



Game Programming with Unity and C#

A Complete Beginner's Guide

Casey Hardman

Apress®

Game Programming with Unity and C#

A Complete Beginner's Guide

Casey Hardman

Apress®

Game Programming with Unity and C#

Casey Hardman
West Palm Beach, FL, USA

ISBN-13 (pbk): 978-1-4842-5655-8
<https://doi.org/10.1007/978-1-4842-5656-5>

ISBN-13 (electronic): 978-1-4842-5656-5

Copyright © 2020 by Casey Hardman

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Development Editor: James Markham
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5655-8. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

Table of Contents

About the Authorxv

About the Technical Reviewerxvii

Introductionxix

Chapter 1: Installation and Setup 1

 Installing Unity 1

 Installing Our Code Editor 4

 Creating a Project 7

 Summary..... 9

Chapter 2: Unity Basics 11

 Windows 11

 Project Window 12

 Scene Window 13

 Hierarchy Window 14

 Inspector Window 14

 Components 14

 Adding GameObjects..... 17

 Summary..... 20

Chapter 3: Manipulating the Scene 21

 Transform Tools..... 21

 Positions and Axes 23

 Making a Floor 24

 Scale and Unit Measurements 24

 Summary..... 26

TABLE OF CONTENTS

Chapter 4: Parents and Their Children..... 29

Child GameObjects..... 29

World vs. Local Coordinates..... 32

A Simple Building..... 33

Pivot Points 36

Summary..... 39

Chapter 5: Prefabs..... 41

Making and Placing Prefabs 41

Editing Prefabs..... 42

Overriding Values 44

Nested Prefabs..... 48

Prefab Variants..... 49

Summary..... 50

Chapter 6: Programming Primer 53

Programming Languages and Syntax 53

What Code Does..... 54

Strong vs. Weak Typing 55

File-Type Extensions 57

Scripts..... 57

Summary..... 59

Chapter 7: Code Blocks and Methods..... 61

Statements and Semicolons 61

Code Blocks 61

Comments..... 63

Methods 64

Calling Methods 67

Basic Data Types..... 69

Returning Values with Methods 69

Declaring Methods..... 70

Operators	73
Summary.....	75
Chapter 8: Conditions	77
The “if” Block.....	77
Overloads	79
Enums	80
The “else” Block	81
The “else if” Block	82
Operators for Conditions	83
Equality Operators	83
Greater Than and Less Than	85
Or.....	85
And	86
Summary.....	86
Chapter 9: Working with Objects.....	87
Classes.....	87
Variables.....	89
Accessing Class Members.....	90
Instance Methods	93
Declaring Constructors	95
Using the Constructor	98
Static Members	99
Summary.....	101
Chapter 10: Working with Scripts	103
Usings and Namespaces.....	104
Script Class	105
Rotating a Transform.....	107
Frames and Seconds	109
Attributes	110
Summary.....	112

TABLE OF CONTENTS

Chapter 11: Inheritance 113

 Inheritance in Action: RPG Items 113

 Declaring Our Classes 115

 Constructor Chaining 118

 Subtypes and Casting 122

 Number Value Types 124

 Type Checking 126

 Virtual Methods 127

 Summary 128

Chapter 12: Debugging 129

 Setting Up the Debugger 129

 Breakpoints 131

 Using Unity’s Documentation 135

 Summary 137

Part I: Obstacle Course 139

Chapter 13: Obstacle Course Design and Outline 141

 Gameplay Overview 142

 Technical Overview 143

 Player Controls 143

 Death and Respawn 145

 Levels 145

 Level Selection 146

 Obstacles 146

 Project Setup 146

 Summary 148

Chapter 14: Player Movement 149

 Player Setup 150

 Materials and Colors 152

 Declaring Our Variables 155

Properties.....	158
Tracking the Velocity	159
Applying the Movement	166
Summary.....	170
Chapter 15: Death and Respawning	173
Enabling and Disabling	174
Death Method.....	177
Respawn Method	178
Summary.....	180
Chapter 16: Basic Hazards	181
Collision Detection	181
Hazard Script	188
Projectile Script.....	190
Shooting Script	194
Summary.....	197
Chapter 17: Walls and Goals.....	199
Walls	199
Goals	201
Build Settings for Scenes.....	204
Summary.....	207
Chapter 18: Patrolling Hazards.....	209
Resembling a Patrol Point.....	209
Arrays.....	210
Setting Up Patrol Points	212
Detecting Patrol Points	216
The “for” Loop	218
Sorting Patrol Points	221
Moving the Patroller.....	226
Summary.....	229

TABLE OF CONTENTS

Chapter 19: Wandering Hazards 233

Wander Regions 233

A Basic Editor Extension 236

Editor Scripts 236

Custom Inspectors..... 237

Accessing the Inspector Target 237

Drawing to the Scene 238

Wanderer Setup 240

Wanderer Script 241

Handling the State 243

Reacting to the State 245

Summary..... 247

Chapter 20: Dashing 249

Dashing Variables 249

Dashing Method 251

Final Touches 254

Dash Cooldown 255

Summary..... 257

Chapter 21: Designing Levels 259

Prefabs and Variants 259

Making Levels 261

Adding Walls 263

Level View Camera..... 264

Summary..... 264

Chapter 22: Menus and UI 267

Scene Flow 267

Level Selection Script 269

Summary..... 276

Chapter 23: In-Game Pause Menu	279
Freezing Time.....	279
Summary.....	284
Chapter 24: Spike Traps.....	285
Designing the Trap	285
Raising and Lowering	289
Writing the Script.....	290
Adding Collisions.....	295
Summary.....	297
Chapter 25: Obstacle Course Conclusion.....	299
Building the Project.....	299
Player Settings	301
Recap	303
Additional Features	304
Summary.....	306
Part II: Tower Defense.....	307
Chapter 26: Tower Defense Design and Outline.....	309
Gameplay Overview	309
Technical Overview	311
Project Setup	313
Summary.....	313
Chapter 27: Camera Movement	315
Setting Up	315
Arrow Key Movement.....	318
Applying Movement	320
Mouse Dragging.....	322
Zooming	324
Summary.....	325

TABLE OF CONTENTS

Chapter 28: Enemies, Towers, and Projectiles..... 327

Layers and Physics 327

Basic Enemies..... 329

Projectiles 333

Targeters 339

Towers..... 348

Arrow Towers 351

Summary..... 359

Chapter 29: Build Mode 361

UI Basics 362

The RectTransform..... 366

Building Our UI 368

Events 373

Setting Up 374

Build Mode Logic 379

The Dictionary 385

OnClick Event Methods 388

Summary..... 395

Chapter 30: Play Mode 397

Spawn and Leak Points..... 397

Locking the Play Button 399

Pathfinding Setup 401

Finding a Path 404

Play Mode Setup 409

Spawning Enemies 414

Enemy Movement 417

Summary..... 423

Chapter 31: More Tower Types	425
Arcing Projectiles	425
Cannon Tower	431
Hot Plates	435
Barricades	437
Summary	437
Chapter 32: Tower Defense Conclusion	439
Inheritance	439
UI	441
Raycasting	441
Pathfinding	442
Additional Features	442
Health Bars	442
Types for Armor and Damage	443
More Complex Pathing	443
Range Indicators	444
Upgrading Towers	445
Summary	445
Part III: Physics Playground	447
Chapter 33: Physics Playground Design and Outline.....	449
Feature Outline.....	449
Camera	449
Player Movement.....	450
Pushing and Pulling.....	450
Moving Platforms	451
Swings.....	451
Force Fields and Jump Pads	451
Project Setup	451
Summary.....	452

TABLE OF CONTENTS

Chapter 34: Mouse-Aimed Camera..... 453

 Player Setup..... 453

 How It Works..... 454

 Script Setup 456

 Hotkeys 463

 Mouse Input..... 464

 First-Person Mode 469

 Third-Person Mode 470

 Testing..... 474

 Summary..... 475

Chapter 35: Advanced 3D Movement..... 477

 How It Works..... 477

 Player Script..... 480

 Movement Velocity..... 486

 Applying Movement 490

 Losing Velocity 493

 Gravity and Jumping 494

 Summary..... 496

Chapter 36: Wall Jumping 497

 Variables 497

 Detecting Walls 499

 Performing the Jump 502

 Summary..... 505

Chapter 37: Pulling and Pushing 507

 Script Setup 507

 FixedUpdate 511

 Target Detection..... 513

 Pulling and Pushing 515

Cursor Drawing	517
Summary.....	518
Chapter 38: Moving Platforms.....	519
Scene Setup.....	520
Platform Movement.....	522
Player Platforming	529
Summary.....	532
Chapter 39: Joints and Swings.....	533
Swing Setup.....	533
Connecting the Joints	540
Finishing Touches.....	542
Summary.....	543
Chapter 40: Force Fields and Jump Pads	545
Script Setup	545
Force Field Setup.....	547
Adding Velocity to the Player	548
Applying Forces	549
Summary.....	552
Chapter 41: Conclusion.....	553
Physics Playground Recap.....	553
Further Learning for Unity.....	555
The Asset Store	555
Terrains.....	555
Coroutines	556
Script Execution Order.....	556
Further Learning for C#.....	557
Delegates.....	557
Documentation Comments	558
Exceptions	560

TABLE OF CONTENTS

Advanced C# 562

 Operator Overloading..... 562

 Conversions..... 562

 Generic Types 563

 Structs 563

Summary..... 563

Index..... 565

About the Author



Casey Hardman is a hobbyist game developer, who found inspiration in the capacity for immersion and interactivity provided by games. His area of focus is the Unity game engine. He has nurtured a passion for video games since he was a child. In his early teens, this interest led him on a journey into the world of game design and programming. He is self-taught through a variety of personal projects, some small and some lofty. He has been a regular contributor on various online game development platforms and spends far too much time in front of the computer.

About the Technical Reviewer



Robert Lair has been building software professionally for more than 25 years and has served at just about every position in the SDLC. He has served as President/CEO, VP of Product Development, CTO, Software Architect, Developer, and Scrum Master/Product Manager. He believes that building good software is an art and is passionate about correctly architected software, efficient development processes, documentation, reducing technical debt, organization, and productivity. He specializes in building Unity, web, and mobile applications focused on gamification. You can find out more about Robert at his website, www.robertlair.com.

Introduction

Welcome to the start of your adventure into game programming with Unity. This book is designed to teach you how to program video games from the ground up, while still engaging you with plenty of hands-on experience. It's not focused on completing ambitious projects, and it's not about fancy graphics. We're learning how to program and how to use the Unity engine. Once you have a solid understanding of these integral topics, you can expand your knowledge and make more and more complicated and impressive games.

All of the software we'll be using is cross-platform. This book will mostly stick to Windows-based terminology and examples, but you can still follow through with other major operating systems, like Mac or Linux, with little or no extra trouble.

As for system requirements, any modern computer purchased within the last 6 years or so should have little difficulty running the software we'll be working with. Since we aren't fiddling with high-end graphics or computing long-winded algorithms, the example projects we develop should run fine on most systems. If you have concerns, the official and most up-to-date system requirements for Unity can be found at the official website here:

<https://unity3d.com/unity/system-requirements>

In Chapters 1–12, we'll begin with a primer for the essential concepts of the Unity game engine itself and get all our tools set up and ready for action. Then, we'll get into the nitty-gritty details of programming, and we'll start to actually write code ourselves.

In the remainder of the book, we'll tackle individual game projects one at a time, making playable projects that you can add to later if you please. This is where you'll get much of your hands-on experience. We'll implement actual game mechanics, which is what you're really after as a game programmer, right?

Game Project 1, “Obstacle Course” (Chapters 13–25), will be a top-down obstacle course where the player moves their character with WASD or the arrow keys to avoid touching hazards of various forms: patrolling and wandering hazards, traveling projectiles, and spike traps in the floor. We'll get practice with basic movement and rotation, setting up levels, working with fundamental Unity concepts like prefabs and scripting, and setting up UI.

INTRODUCTION

Game Project 2, “Tower Defense” (Chapters [26–32](#)), will be the basis of a simple “tower defense” game, where the player places defensive structures on the playing field. Enemies will navigate from one side of the field to the other, and the player’s defenses will attempt to fend them off. We’ll explore basic pathfinding (how the enemies navigate around arbitrary obstacles) and further expand on fundamental programming concepts.

Game Project 3, “Physics Playground” (Chapters [33–41](#)), will be a 3D physics playground with first- and third-person camera support for a player character with more intricate mouse-aimed movement, jumping, wall jumping, and gravity systems. We’ll explore the possibilities of Unity physics, from detecting objects with raycasts to setting up joints and Rigidbodies.

CHAPTER 1

Installation and Setup

Installing software is somewhat simple – download an installer, run the installer, a menu (sometimes called a “wizard”) pops up, you agree to some terms of use, it asks you where you want to install the program on your computer, maybe it offers some additional options, and then it starts installing. Easy, right? So I won’t go into painstaking detail over the installation process. I’ll just show you what to install.

Installing Unity

Unity is frequently releasing new versions with new features, bug fixes, and little improvements. Because of this, they’ve recently come up with what they call Unity Hub. It’s a lightweight little application that lets you install the actual Unity engine, including older versions of the engine. It also lets you manage older versions of the engine already installed on your computer and view all your Unity projects from one place.

Sometimes it’s useful to keep an old version of Unity around even after you upgrade to the latest version. You may want to work on an older project with the same version you started with, in case some new features or changes aren’t compatible with your old project – things change, and sometimes the new stuff breaks the old stuff. Sometimes the old stuff just gets reworked and isn’t valid in a newer engine. In those cases, you might decide to stick to the old version until you finish a project, to avoid spending unnecessary time changing the way you did something to the “new way.”

So we’re going to install the Unity Hub first, and then we can install the Unity engine itself through the Hub. To download the Hub, navigate to this link in your web browser:

<https://unity3d.com/get-unity/download>

Click the button titled “Download Unity Hub,” as shown in Figure 1-1.

A green rectangular button with the text "Download Unity Hub" in white, centered within the button.

Figure 1-1. *Download Unity Hub button*

The Hub installer will begin to download. Run the installer and follow the prompts.

Once the Hub is installed, run it. You may be prevented from getting very far by Unity asking you to accept a license, which may involve creating an account with Unity. This is a little one-time setup that pretty much stays logged in and accounted for afterward. It won't bother you much once it's done – but still, if you're prompted to make an account, don't forget your password and username!

To understand what the “license” is, know that Unity used to have a free version and a Pro version. They restricted some of the features from the free version, and you would have to pay for the Pro version if you wanted to use these features – the fancy stuff, particularly fancy 3D lighting and effects. Then, they simply opened up nearly all the features to the free version, while the paid versions offered mostly miscellaneous things like heightened support, extra resources, and team collaboration tools. Now, they offer three different “licenses” to use with the engine: Personal, Plus, and Pro.

As long as you or the company you represent made less than \$100,000 in gross revenue in the previous year (that's us), then you can use Unity's Personal license, which is free of charge. If your game development career takes off and you start pulling in some money, you'll eventually have to upgrade to Unity Plus (\$25/month) or Unity Pro (\$125/month) to avoid violating the license. But let's not get ahead of ourselves – we're just hobbyists for now, anyway.

You might also be asked to fill out a little survey. It's just some questions pertaining to how you plan on using the engine, where your interests lie, and other “get-to-know-you” stuff like that.

Once you've got a license and an account, you should see an “Installs” tab on the left side of the Hub (see Figure 1-2).

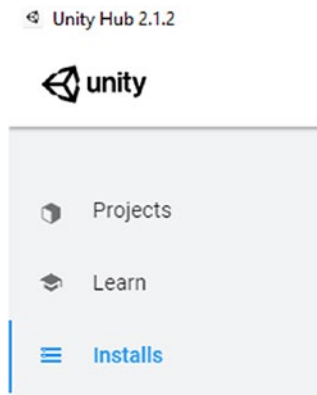


Figure 1-2. Top-left corner of the Unity Hub, with the Installs tab selected

This is where you can see all the versions of the Unity engine you have installed on your computer. You can also install new versions – although installing many versions can quickly use up space on your hard drive, so you may want to uninstall old versions to avoid this. With the “Installs” tab selected, click the “Add” button in the top-right corner. A popup will offer a list of versions you can choose from. The topmost version will be the latest stable release. Click it to select it; then click Next in the bottom-right corner.

You’ll be asked to select what the Unity Hub calls “modules” to install with the engine. These are extra little features to add to the installation, which take up some extra space on your computer if you choose to install them. You can always add modules later, after installation completes, if you find that you want them. Most notable are the Build Support modules, which allow you to build a Unity game project to different operating systems, environments, and hardware.

“Building” a game project is the process of turning it from a Unity project, playable only through the Unity engine, to an actual application used to play the game.

The major platforms that Unity projects can be built to are

- PC, with support for Windows, Mac, and Linux
- Android
- Apple iOS
- WebGL (played in a web browser)
- Xbox One
- PS4

As I said, if you need to build to these platforms down the road (we won't get into them in this book), then you can always install them through the Hub.

You can also select to install the Unity documentation locally as a module. The documentation can be immensely helpful. It's available online as well, and I myself tend to use the online resources, but if you plan on using Unity offline, you may want to install the documentation so it's available without an Internet connection.

For now, you can simply check the Build Support for the operating system you're using, and if you want for the documentation, and uncheck everything else. Then click Done, and the installation will begin. The version you're installing will appear as a box in the main body of the window, and a little bar above it will depict how far along the installation is.

Once it's done installing, the bar will disappear, and you can now create a Unity project with that version of the engine – in a little bit, we'll do that and run the engine for the first time. Another nice feature about the Hub is that it will automatically run the correct editor version of Unity when you open a project (assuming the version is still installed on your computer).

Installing Our Code Editor

You don't write code in the same sort of software that you might write a book or a resume in. Code editors are text editors that are fine-tuned for writing code. They have special highlighting for words and symbols, they know how to format code, and they often come with a slew of features that make it easier and faster for us to write and work with code.

Our code editor of choice is Microsoft Visual Studio Code. It's not to be mixed up with Microsoft Visual Studio. Both are similar (and similarly named) products from the same company, both are free to use, and you could use either one to do the job. They can both edit C# code and integrate with Unity.

Visual Studio Code is designed to be cross-platform and lightweight out of the box, but highly extensible. It has a minimalistic user interface, and most of its features are enabled through installing extensions to add extra functionality, which you do through the software itself.

Visual Studio has support for Windows and macOS, but not Linux. It comes with more features out of the box. It's a very powerful tool and certainly has plenty of uses, including collaboration with teams and other such more advanced features. It is generally "heavier" – more feature-rich but likely to consume more memory and run a little slower.

Code is our choice because I feel it's more suitable for beginners, as it doesn't "get in the way" as much, and most of the advanced features of Visual Studio don't really suit our workflow anyway.

To download Code, head on over to this link in your favorite web browser:

<https://code.visualstudio.com/download>

From there, you can select the correct button to download the software based on your operating system (Windows, Linux, or Mac). The installer should begin to download. Once it completes, run it and follow the instructions it provides.

Once you have Visual Studio Code running for the first time, you'll see a welcome page serving as a hub for various links and resources. Many programmers are picky about the color scheme their code editor uses. I prefer dark schemes myself. Some prefer light – whatever floats your boat. You can easily change it right from the welcome page. Click the "Color theme" button to pop up a list of standard color themes, allowing you to switch to whichever theme you prefer (see Figure 1-3).

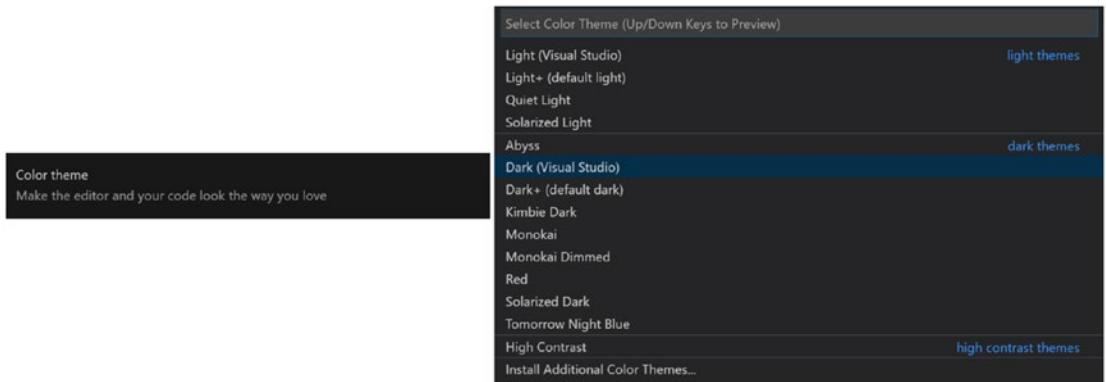


Figure 1-3. Welcome page Color theme button (left) and resulting popup (right)

Once you're satisfied with your colors (you can always change them again later), let's close the welcome page. Any file or page you have open in Code will have a tab at the top left (see Figure 1-4) whether it's an editable code file you're working on or a static page like the welcome page. If you have multiple files open, you can easily switch to view a different one by clicking the tab. Right now, we only have the welcome page open. Let's close it to get a blank slate. You can do this by left-clicking the X button on the tab or by using the Ctrl+W hotkey.

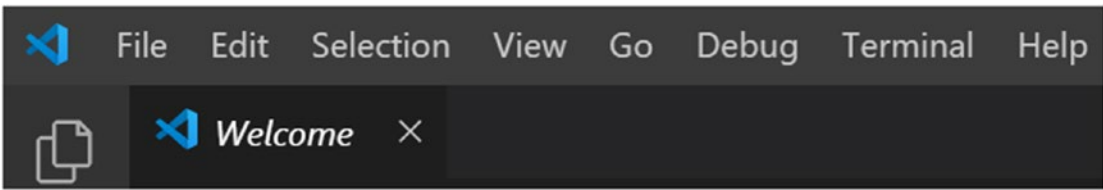


Figure 1-4. The top-left corner of Code, where all page tabs are shown. The welcome page tab is the only one we have open here

Once it's closed, there won't be much going on – just a big blank space, as shown in Figure 1-5.

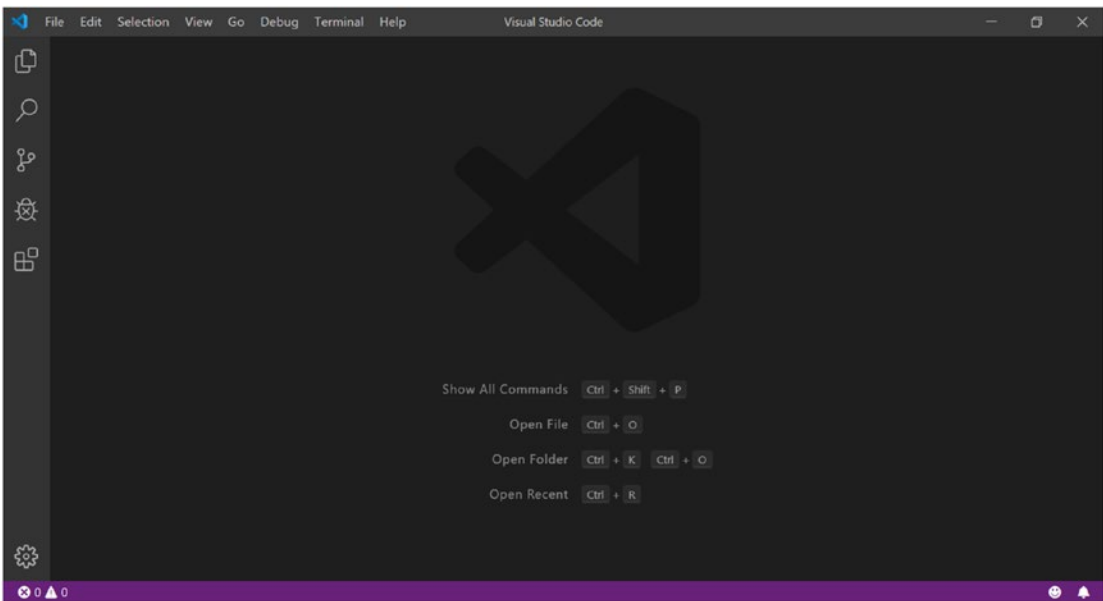


Figure 1-5. View of Visual Studio Code with no files open

You can now install what Code calls “extensions” to add some extra functionality to the editor. Extensions are managed and installed through a button on the left sidebar. The left sidebar has a handful of different forms it can take, all based on which of those buttons you pressed last. Mouse over the buttons to see what they mean and click the Extensions button when you find it (it's the bottom one). You can also press Ctrl+Shift+X to bring it up. This will cause the sidebar to pop up on the left side of the screen. Pressing the button again will fold the sidebar, tucking it away.

Inside the Extensions sidebar, you can search for extensions with the search bar at the top. Click within the search bar and type “C#”. You should see a result simply titled “C#”, with a description “C# for Visual Studio Code.” You’ll also notice that the publisher for the extension is listed beneath the description: “Microsoft,” which so happens to be the company behind Visual Studio Code and C# itself.

Click this extension, and a new tab will pop up, providing details about the extension. Under the main description at the top of the page, you’ll see a button to install the extension. Click that, and the extension will begin installing. If you’re prompted to install any further extensions by popup boxes during this, go ahead and permit them to install.

Next, we’ll install the extension for debugging in Unity. This allows us to “attach” our code editor to Unity, so that we can use the code editor to set up “breakpoints” in the code. A breakpoint is a point in the code that, when reached during the execution of the program, causes the whole program to freeze. While frozen, we can look at pretty much any piece of data from our program that we want and resume whenever we please, among other things. It’s a very handy feature that we’ll use later down the road.

The exact name of the extension is “Debugger for Unity,” published by Unity Technologies. You can find it the same way – by typing the name in the search bar at the top of the Extensions sidebar.

Once you have these extensions installed, we can close Visual Studio Code. We’ll be using it more in Part 2. For now, we’re focused on the Unity editor itself.

Creating a Project

We can now use the Unity Hub to create our first project, so we have an environment to play around in as we learn. In the Unity Hub, click the Projects tab on the left side, and then click the blue “New” button in the top-right corner.

A dialog box will appear (see Figure 1-6), allowing you to select a template to base the project on.

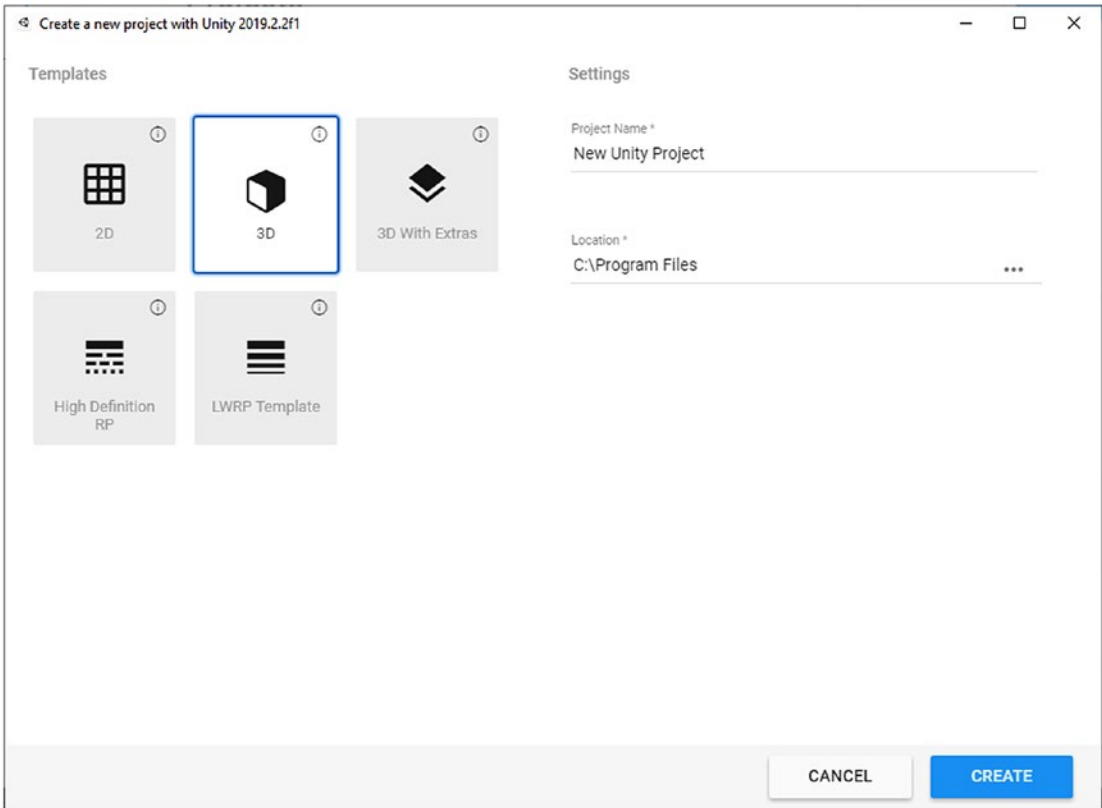


Figure 1-6. *Dialog box for creating a new Unity project*

The template is just a simple starting point for your project. The most bare-bones and simple templates are the first two, titled “2D” and “3D.” These are pretty much blank slates, one set up for 2D and one for 3D.

We’ll start with a blank 3D project, so select the 3D template by clicking it. It will have a blue border around it if it is selected.

To the right side of the template options, there’s a field for the project name (you can change this later) and the directory (file path) on your computer to save the project files to. We’ll name our project “ExampleProject” – note that we aren’t using a space between the two words, because file paths aren’t always fond of them.

You can change the directory to save the project wherever you like. Whatever file path you choose, a folder named after the project will be created inside that folder. That folder is the “root directory” for your project, and all your project’s files and resources will be stored in that folder.

Once you’ve selected the path you want, click the blue “Create” button in the bottom-right corner, and wait for Unity to create the base project files. When Unity finishes, the editor itself will pop up with your brand-new project opened and ready for editing.

Summary

The following is a recap on what we learned in this chapter:

- The Unity Hub program will be used to download new versions of the Unity editor, uninstall old versions you no longer need, create new projects, and open existing projects.
- Opening a project in the Unity Hub will start the Unity editor, which is where we’ll actually use the engine to develop our game.
- A Unity game project is stored on your computer, with all the related files – including stuff we make ourselves like art and code – stored in a “root directory” named after the project name.
- Our code will be written with Visual Studio Code, a text editor designed specifically for writing code. It will offer us useful features that normal text editors don’t have, making it easier to format and navigate our code.

CHAPTER 2

Unity Basics

Now that we have Unity set up and a new project to work with, let's get comfortable with the engine. This is the user interface we will be interacting with quite a lot as we develop our games, after all, so we ought to get to know it early.

Windows

Unity is separated into different windows that serve different purposes. Each window has a little tab at its top-left corner, where the name of the window is written – that is, what type of window it is.

Much like in a lot of computer software nowadays, each of these windows is a separate piece of the program that can be repositioned, resized, or even altogether removed. There are a handful of other window types that aren't being used now, so we don't see them on our screen – but if we ever wanted to use them, we could add them in a jiffy.

You'll notice that if you left-click one of these window tabs, hold the mouse button down, and drag it, the window can be picked up and moved to a different spot in the program. Doing this, you can split one window's space to cover some of it with a different window. If you ever want to do something with a window, chances are you just need to start dragging stuff around and see how Unity reacts.

You can also dock windows beside others to place their tabs side by side (see Figure 2-1). Whichever tab you clicked last will gain focus and fill the space of the window.

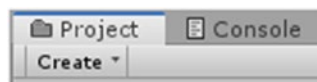


Figure 2-1. Two window tabs docked side by side in the same space. The Project window has focus and will fill the space, but the Console window can be given focus instead by clicking its tab

Unity also lets you save all your current windows and their sizes and positions in a **layout**, which you can assign a name to. This is done with the little dropdown button in the top-right corner of the Unity editor, with the text “Layout” written on it. Click this button to see all of the built-in layouts you can choose from, as shown in Figure 2-2.

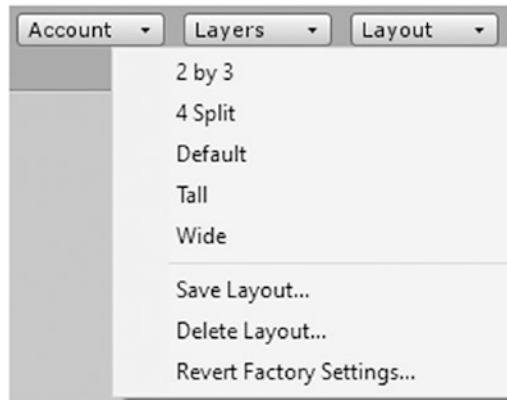


Figure 2-2. The Layout button in the top-right corner of the Unity editor, after it has been clicked to show its options

By default, it will be set to a layout very aptly named “Default.” Try clicking a different layout, and you’ll see Unity automatically restructure all its windows for you. You can also select the “Save Layout...” option in the dropdown list to save your current layout under a new name to keep it around, once you’ve set things up the way you like. That way, if you ever lose a window by accident or accidentally cause some catastrophe to your window setup, you can just reset it to the way you had it by loading your layout with the dropdown.

Layouts can also be useful when dealing with different aspects of game development. Certain activities might be easier to do with a different set of windows in different positions. The ability to save layouts makes it easier for us to hop back and forth between activities – we can do it with just a few clicks.

The default layout has all the most important windows. Let’s go over them to learn about what they do.

Project Window

The Project window shows us all our **assets**. An asset is the game development term for some piece of work we can use in our game – art, sound effects, music, code files, game levels, things like that.

Once we get around to making assets, we'll see them here in the Project window. It pretty much functions like your computer's file system. You store assets in folders, also known as directories. You might have a folder for all your sound effects, another for all your code files (called "scripts"), and so on. We organize it all however we like in this window, and it's from this window that we can find, select, and use our assets within the engine.

Our project has been set up with some folders for us by default: the Packages folder which we don't need to pay any attention to right now and the Assets folder which is the root folder inside which all our assets will be stored. The arrow beside a folder can be clicked to hide or show its contents (if it has anything inside it). If you unfold the Assets folder, you'll see it already has a Scenes folder inside it and an asset "SampleScene."

Scene Window

The Scene window lets you see the environment your game is taking place in. What we might call a "level" in our game is a "scene" in Unity. Scenes are saved as assets, so we'll see them in our Project window when we save them – this scene is the "SampleScene" asset we have in our project by default.

Each scene has its own collection of objects in it. The scene window lets you view a scene and navigate through it, like a floating camera observing your game world. It's your main viewport into the game environment.

The sample scene we have open now doesn't really have anything inside it yet – just a light, which is an invisible object that casts light over everything in the scene, and a camera, which is the object through which the player will see the scene when the game is being played.

If we had other scenes, which we will later, we would store them all in our Scenes folder, and double-clicking one of them there would load a different scene, letting us view and edit that scene instead.

Within the Scene window, moving your mouse while holding right-click will turn the camera, much like looking around in a first-person game. While holding right-click, you can use the WASD keys to move the camera around – once again, much like in a game. W to move forward, S to move backward, A to move left, and D to move right. You can also use Q to move directly down and E to move directly up.

Hierarchy Window

The Hierarchy window allows you to see the objects contained within the current scene. As we just said, a scene is pretty much just a collection of objects. When we switch from one scene to the other, we're just tucking away all the objects in the current scene and pulling out all the objects in the new scene instead.

You can see those two objects we mentioned in our scene are listed in the Hierarchy: a “Directional Light” and a “Main Camera.” By default, we'll have a light source and a camera in our scene, so we'll see them all listed here in the Hierarchy.

These are **GameObjects**. Simply put, a GameObject is some object in your scene. It could be a prop, like a box or a plant or a tree. It could be the player character or an enemy or a powerup on the ground. They can be a disembodied light or a GameObject that does nothing; it just exists, invisible in the scene. At their simplest, they're just a point in space with a name.

Inspector Window

The Inspector window is a very important window that we will use extensively in our adventures with Unity.

As we just went over, the Hierarchy window shows a list of all GameObjects in the scene. Click one of these GameObjects in the Hierarchy window to select it. You will notice that this causes the Inspector to change. It's now showing you information about the selected GameObject. At the top of the Inspector, you can see a box containing the GameObject name, which you can click to type a new name in if you please. There are also a few dropdown buttons, one for the “tag” and one for the “layer” – we'll learn what those are for later.

But the main functionality of the Inspector is to show us all the **components** attached to the selected GameObject(s).

Components

A component is a Unity term for a feature, some single piece of game functionality, that is attached to a GameObject. A component cannot exist without a GameObject to attach it to.

There are many different kinds of components that serve different purposes, all included in the Unity engine by default. There's the Light component to cast light, whether it's a sun-like light that covers the whole scene or something like the beam of a flashlight.

Let's look at a Light component. Since the Inspector is designed to show us the components of the selected GameObject, try clicking the Directional Light in the Hierarchy window to select it. You'll see the Inspector update after you do this, to look something like Figure 2-3.

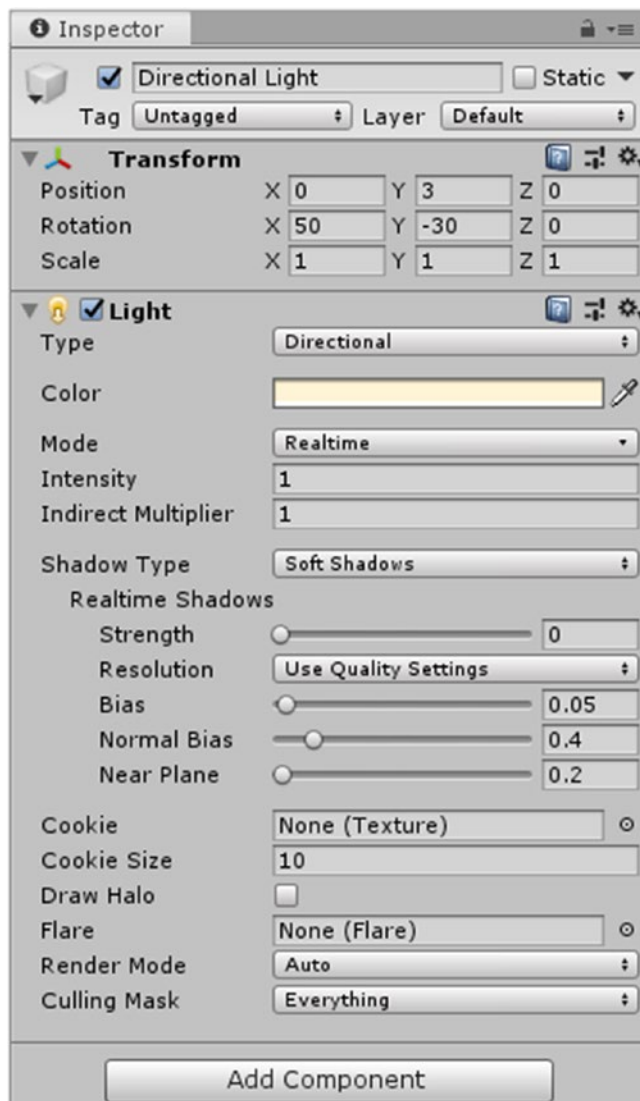


Figure 2-3. A view of the Inspector with the default Directional Light GameObject selected

Beneath the basic information at the top, like the `GameObject`'s name, you'll see a heading for each component attached to the `GameObject`. This `GameObject` has two components attached: `Transform` and `Light`. You can click the headings of these components (where the component name is) to hide them (also known as "folding" them) or show them. Beneath the header, the various properties of the component are listed as value fields that we can change to affect the component – make the light more intense, change its color, change the way it casts shadows, and so on. The field name is on the left side, and the field value is on the right side. Some of these field values are numbers, and some are little "sliders" that let you click and drag a dial with your mouse – you'll come across many kinds of fields designed to edit different sorts of values. This is the major functionality of the Inspector: viewing and editing the properties of components on a `GameObject`.

Another example is the `Camera` component. It's an essential. It's what renders ("render" means "draw to the screen") the scene to the player's screen when the game is playing. If you select the Main Camera `GameObject` in the Hierarchy window, you'll see the `Camera` component listed in the Inspector.

When we start making things ourselves later in the book, we'll be getting some hands-on experience with using many kinds of components. Unity also has helpful official documentation, and this can be easily accessed by clicking the little icon that looks like a book with a ? symbol on the cover, located on the right side of each component header in the Inspector. This will open the documentation for that specific component type in your default web browser.

Our code will be attached to `GameObjects` in the form of components as well. They're called "scripts" in this case: that is to say, a script is a component which runs our code when it's attached to a `GameObject`. So when we write code, the way we get it into the game is by attaching it as a script component to a `GameObject`.

This means we can reuse and mix different pieces of functionality, if we're smart about how we write and define our scripts.

For example, our first example project is a game where the player must avoid various obstacles. Let's say you're making a project like this, and you want to make different kinds of obstacles to keep the game interesting. You want obstacles that shoot fireballs, blades that spin in a circle, and rolling spike balls that move back and forth between two points.

Each of these pieces of functionality can be made into a separate script component: Shooting, which periodically fires projectiles in front of the `GameObject`; Spinning,

which makes the object constantly twirl around; and Patrolling, which makes the object pace back and forth between two or more points. Then, we can have a component called Hazard that we attach to fireballs, the spinning blades, and the patrolling spike balls, which makes them kill the player when they touch.

The cool thing about components is that we can then mix them with each other to create new types of obstacles.

Because each different piece of functionality is contained within its own component, we can make anything shoot, spin, patrol, or kill the player on touch. And since we aren't limited in how many components we add to a single `GameObject`, we can make a fireball-shooter that spins in a circle by attaching both the Shooting and Spinning components to a single `GameObject`. We can attach a blade to its opposite side which acts as a hazard. We can make a patrolling spike ball that shoots fireballs in front of it.

You get the point. As long as each piece of functionality is part of its own script component, we can simply throw any combination of script components onto one `GameObject` to fuse all the different things we coded onto one object.

This is one of the major advantages of Unity's component system. It provides a building block sort of system where we can mix different features however we please.

Adding GameObjects

Let's start getting familiar with using Unity to create and manipulate some `GameObjects`. We won't be making character models, spaceships, guns, or anything fancy like that from within the Unity engine. Unity is not a modeling package (to create 3D objects) or an image editor (to create 2D objects). It's a game engine; you make the models and animate them in other software, and then you import it into Unity by simply putting it into your project folder, and Unity makes sense of it and lets you drag and drop it into your scenes.

For the sake of learning the engine and learning to program, we won't be messing around with fancy art.

However, Unity can create `GameObjects` of basic shapes for us on the fly with just a few button presses.

Just beneath the top of the Unity editor (the title bar), you'll see a bar with a selection of different buttons: File, Edit, Assets, `GameObject`, Component, Window, and Help. These can be clicked to drop down a menu of further options.

The GameObject dropdown menu can be used to create simple, frequently used GameObjects: shapes, cameras, lights, and so on.

Using this menu, we can create a cube through GameObject ► 3D Object ► Cube. Alternatively, you can also right-click anywhere in the Hierarchy, but not on the name of an existing GameObject, and use the context menu to select 3D Object ► Cube, as shown in Figure 2-4.

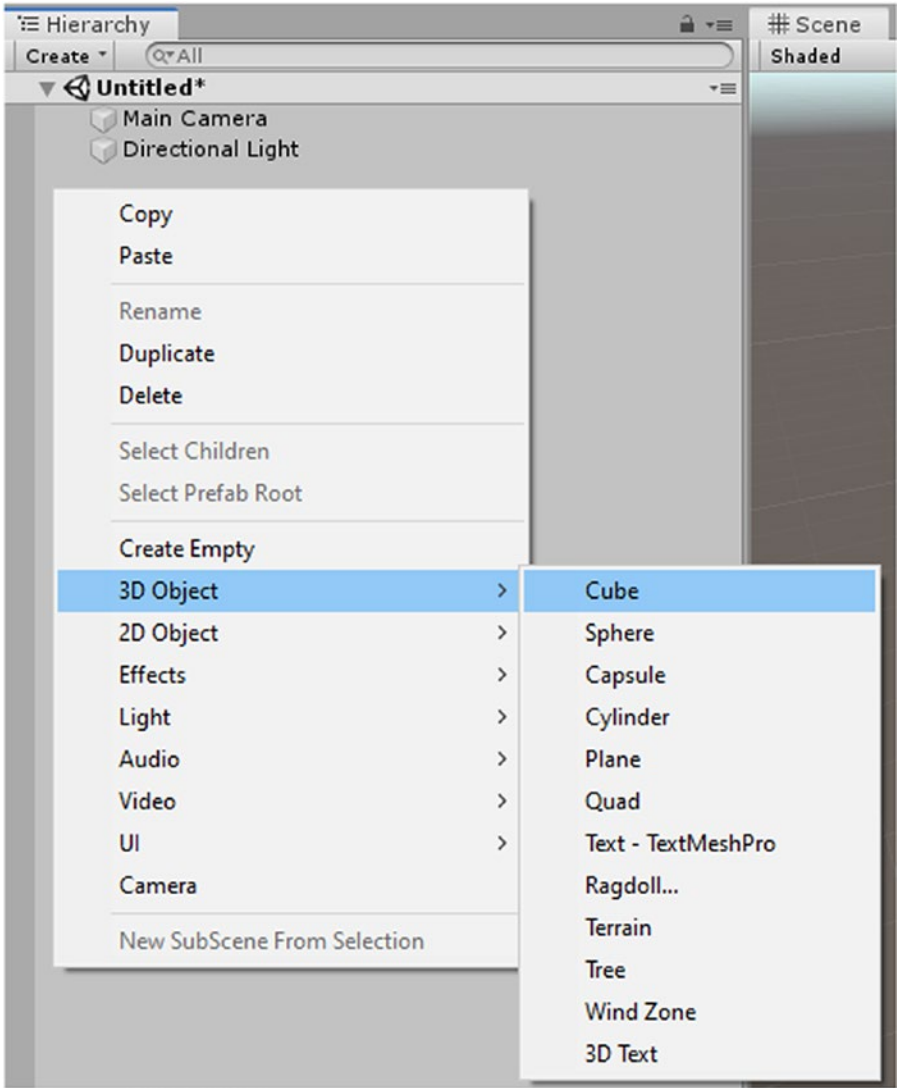


Figure 2-4. Creating a Cube by right-clicking in the Hierarchy and navigating the resulting context menu

After doing either of these things, you'll notice a Cube will be added to our Hierarchy and will show up in the Scene if your camera is pointing at it.

If you can't see it in the Scene, your view is probably out of whack. You can easily navigate your view to a GameObject in the current Scene by clicking the object in the Hierarchy window to select it, then putting your mouse over the Scene window so it has focus, and then pressing the F key. This is a handy shortcut that moves your view over to the selected objects so you can see them. If you ever move so far away from your objects in the Scene view that you get lost, just use this.

To expand on our talk about components earlier, let's check what components are present in this cube we just made. Making sure the cube is selected, look at your Inspector window.

The topmost component will always be a **Transform**. Every GameObject has a Transform component. A Transform is a position, size, and rotation. It's essential to a GameObject. You can remove components on the fly with code, but you can't remove a Transform. You have to delete the whole GameObject if you want to destroy its Transform component. Every object that exists in your scene must have a position, right? It has to *be somewhere*.

Aside from the Transform, you'll notice a few other components.

There's a Mesh Renderer and a Mesh Filter.

The word **mesh** is pretty much synonymous with "3D model" as far as we're concerned. And **render** is just a fancier-sounding word for drawing or displaying something on the screen.

So a Mesh Renderer is a component that allows a 3D model to be drawn, although the drawing of it is done by a Camera component.

The Mesh Filter is a component that holds the mesh you want to pass to the Mesh Renderer. Pretty much whenever you see a Mesh Renderer, you'll see a Mesh Filter as well, because the Filter tells the Renderer what to render.

You'll notice a little checkmark beside the Mesh Renderer component in the Inspector. Components like this can be enabled and disabled by clicking this checkmark. Just in case you don't believe that the Renderer component is actually drawing the cube to the scene, try clicking that checkmark to uncheck it. You'll notice that the cube stops rendering to the screen in the Scene view. Check it again, and it'll pop back up.

Summary

In this chapter, we learned the following:

- The Unity editor is made up of various **windows** that serve unique purposes. Any window can be rearranged and resized by clicking and dragging the tab at its left-top corner.
- An **asset** is a file for use in our game, such as artwork, audio, or code. These are viewed in the **Project window**, and often we'll incorporate them into our game by simply dragging and dropping from that window.
- A **scene** is an **asset** that resembles a game environment, like an individual level. We can load them to view and edit them in the **Scene window**.
- A **GameObject** is an object that exists in the scene. Their functionality is driven by **components** that we attach to them. Unity provides many built-in components for fundamental things like displaying a 3D model, casting light, having physics and collisions, and so on.
- The components attached to an individual GameObject are viewed in the **Inspector window**. Here, we can customize their functionality by editing fields that relate to them – such as how bright a light is. Each component is a unique instance with its own values associated with it, and these values can be customized to provide different functionality.
- Every GameObject has a **Transform**, a basic component that resembles a **location**, a **rotation**, and a **size**. Other component types can be added and removed on the fly through code, but the Transform cannot – there can only be one per GameObject and it cannot be deleted.

CHAPTER 3

Manipulating the Scene

We've learned the basics of the most important windows in the Unity engine, and we know how to create simple objects and view their components through the Inspector. Now let's get familiar with moving, rotating, and sizing GameObjects in our scene.

Transform Tools

The section of the Unity editor just beneath the title bar and the title bar buttons (like File, Edit, Assets, etc.) is called the **toolbar**. It's a bar stretching across the width of the screen, with a handful of different buttons on it. This includes the Layout dropdown menu we learned about earlier, which is the rightmost button in the toolbar.

We're going to learn about that cluster of buttons all the way on the left of the toolbar, shown in Figure 3-1.



Figure 3-1. Buttons corresponding to the transform tools. The second button is currently selected, giving it a darker background than the rest

These are the **transform tools**. You just learned that the Transform component is position, rotation, and size, so you can probably guess that the transform tools are primarily used to move, rotate, and size GameObjects in the scene.

There are six buttons for six different kinds of tools. You may also see a seventh button on the right side that deals with custom editor tools, which we don't need to concern ourselves with right now anyway. If you don't see it, don't worry about that.

Each of these buttons can be clicked to switch to a different tool. Only one tool is ever active at a time, and they all serve different purposes.

From left to right, you can use the hotkeys Q, W, E, R, T, and Y to toggle between these tools, which is often faster than clicking.

The first tool, with the hotkey **Q**, is the **hand tool**, which lets you left-click and drag on the screen in the Scene view to drag your scene camera around. It doesn't edit the scene. It just helps you navigate it.

The other tools will allow you to edit the GameObjects you are selecting. In the Scene window, the transform tool you have selected will provide little "gizmos" on or around your selected GameObjects. These gizmos are simple tools that we click and drag to use the transform tool to interact with the GameObject. You will notice, if you select a GameObject and toggle between these tools, that the gizmo drawn around the object changes as the selected tool changes.

W is the **position tool**. While active, it shows arrow gizmos on your selected GameObject. You can drag the object in specific directions by clicking and dragging the arrows. You can also drag it along two directions at once by clicking the square shapes between arrows. Holding the Ctrl key while dragging will only move in increments of 1 unit at a time.

E is the **rotation tool**. It shows circle gizmos on the selected GameObject. Clicking and dragging the circles will spin the object, and each circle turns it along different directions. You can also click between the circles to turn the object in multiple directions at once.

R is the **scale tool**. It shows gizmos like the arrows, but with cube-shaped ends. Click and drag these boxy arrows to change an object's width (red), length (blue), or height (green). Click the cube in the center of the gizmo and drag to scale the entire object at once – that is, raising or lowering the width, length, and height evenly at once.

T is the **rect tool** ("rect" being short for rectangle). It is most applicable to 2D projects but can have its uses in 3D as well. The gizmo shows a rectangle around the selected object, with circles at the corners. The edges or corners can be clicked and dragged to expand or shrink the object as a rectangle, affecting both the position and scale at once. This can be useful to make an object larger or smaller on one side only, since the scale tool will affect the scale on both sides.

There's also a circle at the center of the gizmo which can be clicked and dragged to reposition the object along the two axes that the rect is aligned with. You'll notice that the rectangle gizmo operates on two axes at any time. Attempting to move your camera over to the side of the rectangle will cause it to flip around and face the camera again.

The Y tool combines the W, E, and R tools, showing the arrows for moving, the circles for rotating, and the cube at the center for scaling, all at once.

Positions and Axes

So how does positioning work in 3D space? It might take a little getting used to, but a position in 3D space is defined by three number values, referred to as X, Y, and Z.

The **X** position is **right and left**.

The **Y** position is **up and down**.

The **Z** position is **forward and back**.

These positions are often written as (X, Y, Z). For example, (15, 20, 25) would be an X value of 15, a Y value of 20, and a Z value of 25.

If you have a position of (0, 0, 0), you are at the “world origin,” so to speak – the center of the universe, or at least the center of the scene.

Add 5 to your X position, and you’ve moved 5 units to the right.

Subtract 5 from your X position, and you’ve moved 5 units to the left.

It works similarly for the Y and Z values: adding moves in one direction, subtracting moves in the opposite.

Adding to the Y will take you up, and decreasing it will take you down.

Adding to the Z will take you forward, and decreasing it will take you backward.

It is the combination of these three values which defines where something is in the world. Each of these is called an axis (plural “axes”). So you might hear people say “the X axis” or “the Y axis” or “the X and Z axes.”

The scale and rotation work much the same way: they have the same three axes, each one determining a different direction.

The **X scale** is the **width** – left and right.

The **Y scale** is the **height** – up and down.

The **Z scale** is the **length** – forward and back.

I’m sure you can imagine how rotation works pretty much the same way. The object’s orientation is defined by three angle values between 0 and 360, which determine how it is turned on each axis.

You’ll notice that the tools we use to position, rotate, and scale objects (W, E, and R, respectively) are all color-coded.

The X axis is always red, the Y axis is always green, and the Z axis is always blue.

This is pretty much universally accepted. Get into making 3D models, and you’ll see the same thing – although some programs consider the Y axis to be forward and back and the Z axis to be up and down, which is opposite to how Unity does it.

Making a Floor

Let's use what we've learned to make some cubes, position them, and scale them. But first, let's make a floor.

Create a Plane, using the same method we made the cube with earlier: `GameObject` ➤ `3D Object` ➤ `Plane`.

A plane is like one surface of a cube – a paper-thin, flat surface that has no thickness. They're one-sided: you can't see them at all if you look at them from the backside. Try navigating your camera beneath the plane and looking up at it. You won't see anything, as if it never existed. Still, it'll serve fine for our floor, because we don't expect to be looking at it from below.

Because we know exactly where we want our floor to be, we can set it up using the Inspector. With the new Plane selected, look to its Transform component in the Inspector.

As stated before, the Transform has a position (where it is), rotation (how it's turned about), and scale (how big or small it is).

Remember, the Inspector's primary purpose is to interact with components, not just to view their data. So it exposes the actual values of the position, rotation, and scale of the Transform to us. We can edit the individual axes to our liking, simply by clicking these fields and typing in the numbers we want.

This is a useful way to set things up if you know exactly how you want to set them up, because getting precise values for positions and rotations using the transform tools can be very tedious. We want our plane to be at the world origin (the center of the scene), so use the Inspector to change its position to 0 on all three axes, if it isn't already. As for the rotation, it should be (0, 0, 0) already, so leave it as is.

Scale and Unit Measurements

Now for the scale. "What is a unit of space?" you might be asking. What does it actually mean when we change a `GameObject`'s position from 0 to 1? How much space is that?

This is a slightly confusing concept for some. You probably expect a straight answer. The Unity developers decided what it is, right? It's a foot, or maybe a meter – perhaps a yard.

But that's not how it works. Don't worry, though; it's still quite as simple. It's just that we must decide what a unit is ourselves. Let's say we decide that 1 unit is 1 foot. So be it.

As long as we follow this in every measurement we make, then that's what 1 unit means. We make our people between 5 and 6 units tall, roughly. If we want something to be just one inch, we make it a twelfth of a unit (roughly .083). If we want something to be a yard, we make it 3 units.

But one more thing we need to note about the scale of a Transform is the scale is not "how many units wide, long, and tall something is." It's actually a **multiplier**. It multiplies the size of the mesh (the 3D model).

The mesh itself has its own size, and then the scale value of the Transform just multiplies that size.

This is fine for a cube. The cube mesh is 1 unit wide, tall, and long. So if we set its scale to 5, it's going to be 5 times 1 on each axis, so it's still just 5 units large.

But a plane is trickier. The mesh itself is 10 units wide and long (and it's paper-thin, so it really has no height). So when we have a scale of (1, 1, 1) with a plane mesh, it's actually 10 units wide and long already.

You can see this if you create a Plane and a Cube, leave the scale of each one at the default value of (1, 1, 1), and position them both in the same place. Note how much larger the Plane is than the Cube, as shown in Figure 3-2.

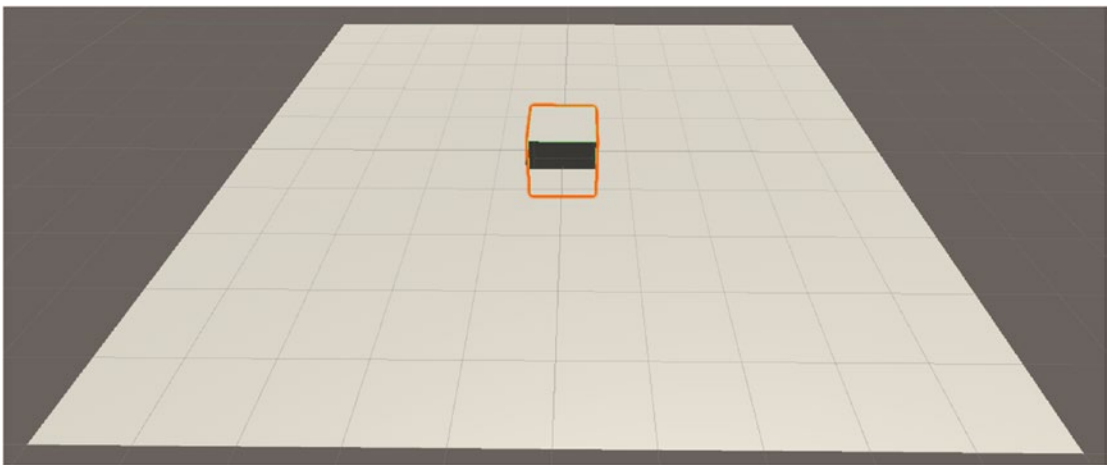


Figure 3-2. *A Plane and a Cube with scale (1, 1, 1) positioned in the exact same spot*

This is because the size of their actual meshes is not the same. Even if their scale is the same, the cube mesh is 1 unit wide and long, while the plane is 10 units wide and long. Since their scale is just a multiplier of the mesh size, not a depiction of the actual

size of the object, the scale of (1, 1, 1) is not affecting the mesh size at all. It's multiplying by 1, so of course, it's leaving them as is.

Now, if we change that cube to have a scale value of (10, 1, 1), it will become 10 units wide, which matches the plane exactly, as shown in Figure 3-3.

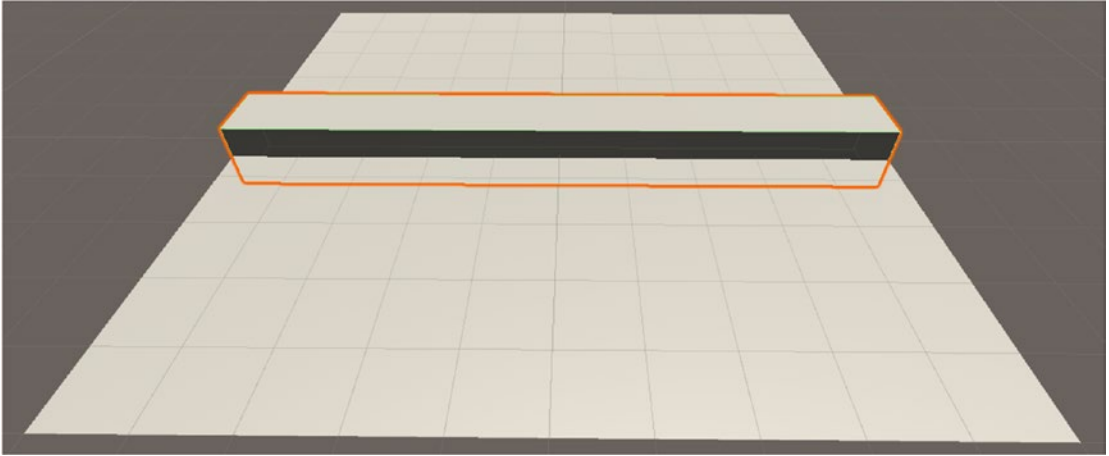


Figure 3-3. A Plane with scale (1, 1, 1) at the same position as a cube with scale (10, 1, 1). The cube is exactly as wide as the plane

To sum it up, just remember this: the mesh has its own size, and the scale is merely a **multiplier** for the mesh size. It is **not** a flat value depicting “how big the mesh is.”

All that aside, let's press on. We want a large floor so we don't have to worry about making it bigger every time we want to put some more stuff on it. Let's make it 10 and 10 on the X and Z scales – which, mind you, is 100 units wide and 100 units long. Of course, it'll be easier to just set the X and Z scale values to 10 in the Inspector rather than using the scaling tool.

Summary

In this chapter, we learned the following:

- How to manipulate the position, rotation, and scale of GameObjects using the transform tools (hotkeys W, E, and R).

- Positions are resembled as an X, Y, and Z value. Adding to a value moves in one direction, while subtracting from it moves in the opposite direction. X is right (positive) and left (negative), Y is up (positive) and down (negative), and Z is forward (positive) and back (negative). By combining all three values, we can define a 3D point in space.
- Scale is a multiplier for the size of the actual mesh that the GameObject is rendering. It's not the number of units wide, tall, and long a GameObject is. Rather, the X, Y, and Z scale of the Transform multiplies the size of the mesh that's being rendered.
- A single unit doesn't correspond to a particular number of feet or inches or meters by default in Unity. We have to decide what a unit means ourselves, and as long as we stay consistent with that decision, our objects will end up properly sized proportionate to each other.

CHAPTER 4

Parents and Their Children

Now that we have our floor set up, let's dive into some very important concepts employed by game engines. Unity allows us to attach individual GameObjects to each other in a system known as “parenting,” where “children” are attached to a “parent” GameObject and thus move, rotate, and scale with it. This creates a distinction between two ways of looking at an object's position – its **world position**, which resembles where it is in the scene, and its **local position**, which resembles where it is in relation to its parent. This also gives us an option to define parents and children in a way that lets us set up **pivot points** to change the point around which objects rotate.

Child GameObjects

There's a reason why the Hierarchy window is called a hierarchy. We haven't exhibited that reason yet, but we're about to.

A GameObject is capable of storing any number of other GameObjects “inside it.” This system is called **parenting**: one GameObject may be the **parent** of many others, and those GameObjects stored inside it would be called its **children**.

Technically, Unity looks at this as the Transform components being attached to each other, because that's where the actual relevance of the concept is seen. Making one GameObject the child of another is like physically attaching it to its parent. Because the Transform deals with the physical position, rotation, and size of an object, this essentially means attaching the Transforms to each other.

When the parent Transform moves, its children move with it. When the parent rotates, the children pivot as if they were attached to the parent, even if there are miles of open air between them. When the parent becomes smaller or larger, the children follow suit proportionately.

Let's play with this concept so you can see it in action.

Create two cubes and position them anywhere apart from each other. It doesn't really matter where, as long as they aren't directly on top of each other – they can touch or overlap if you want. Use the position transform tool (hotkey W) to do this.

Now, scale one cube up a little. You can do so in the Inspector by setting its scale to something like 1.5 on all axes or simply puff it up a little with the scale transform tool (hotkey R). When you're done, it should look something like Figure 4-1.

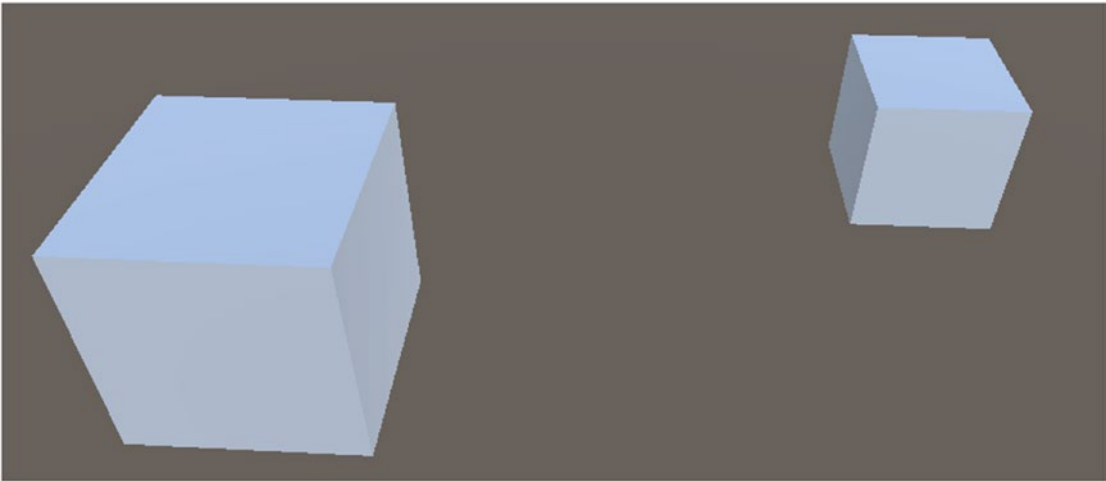


Figure 4-1. A cube with (1.5, 1.5, 1.5) scale (left) and a second, unchanged cube positioned off to its side (right)

That larger cube will be our parent. Select it to see which cube it is in the Hierarchy (it will be highlighted, since it is selected). Using the Inspector, change its name to “Parent” so it’s easy to recognize.

Now, click the other cube in the Hierarchy to select it. Then, click and drag this cube in the Hierarchy, and drop it over the “Parent” cube.

Once you’ve dropped it, the Hierarchy begins to look like a hierarchy for the first time. The dragged cube is now a child of the “Parent” cube and is now “inside” its parent in the Hierarchy. This is shown by its indention: the children will be offset to the right a little further than their parent GameObject. In Figure 4-2, you can see our Parent cube, a Cube child inside it, and another GameObject beneath which is not a child. Notice that the “Cube” child is pushed to the right side a little bit, denoting that it is a child of “Parent.”



Figure 4-2. View of our two cubes in the Hierarchy window. The “Cube” GameObject is a child of the “Parent” GameObject. The third GameObject is not a child, as seen by its indentation

You’ll also notice that the parent now has a little arrow icon on its left side, which can be clicked to hide or show its child GameObjects in the Hierarchy. This can be useful to tuck away a complicated hierarchy of GameObjects that’s cluttering your view.

Now that the cube is a child of the parent, any Transform changes we make to the parent will affect the child cube. The opposite is not true – the child can be moved about to position it differently in relation to the parent, but any movement, rotation, or scaling you apply to the parent will always affect the child as well.

Try it yourself. Select the Parent cube, go to the Scene view, and use the transform tools (W, E, and R) to move, rotate, and scale the cube. When moving and rotating the parent, the child will move with it and pivot around it, as if some invisible bar is attaching them to each other. When scaling the parent, the child will scale as well – they will both grow larger or smaller, and the distance between them is kept the same proportionately to their new size.

We aren’t limited to a simple parent-child relationship. We can have grandchildren, great-grandchildren, and so on – as many layers as we want. We could add another cube and make it a child of the child cube – which would make it a grandchild of the cube we named “Parent.” Then, moving the Parent cube would move its child, and as a result, it would also move the grandchild.

If you ever want to “unparent” a child (make it no longer have a parent), you can do so in the Hierarchy window by clicking and dragging the child above or beneath the parent object. Note that while you’re dragging a GameObject around in the Hierarchy, any other GameObject that you pass your mouse cursor over will be highlighted in light blue. This means you’ll be assigning the dragged object (the one you’re “picking up”) as a child to that highlighted object. If there is no object directly under your cursor, you’ll instead get a thin blue line with a circle at its left side. This means you’re just putting the GameObject somewhere else. You can use this to move the object up or down among its siblings if it has any (siblings are fellow children of the same parent). But you can also use it to put a child at the same level of the parent, making it no longer a child. Pay attention to the left side of the line: that circle will indent further to the right when

placing a `GameObject` within the children of another. This behavior can be a lot easier to observe when you have more complicated hierarchies of `GameObjects`, with multiple levels of children. For our purposes, you can pretty much just drag an object into the empty space below the others and release to make it have no parent.

World vs. Local Coordinates

Let's learn the distinction between world coordinates and local coordinates. The position of an object that has a parent can be depicted in two ways: world position and local position.

The **world position** is its **absolute position in the scene**.

The **local position** is its **position relative to its parent**.

When an object has no parent, the Inspector will show us its world position. Once we give it a parent, the Inspector will instead show us its local position.

A position of (5, 0, 0) in world coordinates is "5 units to the right of the center of the world."

But in local coordinates, that same position instead means "5 units to the right side of the parent object." And that corresponds to the parent's rotation: notice that I wrote "to the right side of the parent," not "to the right." Rotate the parent, and the local position of the child doesn't change. Adding 1 point to the X axis doesn't move the object 1 unit "to the right" in world space – it moves the object 1 unit to the right of the parent object.

This is why even a direction can use local coordinates instead of world coordinates. If you're shooting out a projectile from the player character in a game where the player is rotating around, such as pointing their character with their mouse, then you would use the player character's local forward direction, not the world forward direction.

A good way to look at it is that the world directions are something like directions on a compass. You could consider world forward to be north (positive along the Z axis), world backward to be south (negative along the Z axis), world right to be east (positive along the X axis), and world left to be west (negative along the X axis). A projectile coming out of a gun or the flaming hands of a magician shouldn't just "go north," right? It should go forward along the local direction that the gun or the hands are pointing.

Now let's dig into another complication of local positions: they're affected by the scale of the parent. So if the parent doesn't have (1, 1, 1) for its scale, then adding 1 unit isn't actually adding 1 unit, so to speak. The **local position** is **multiplied by the scale of the parent**. For example, if your parent has an X scale of 2, then adding 1 unit to the X is really adding 2 units in world space.

A Simple Building

Let's use cubes to assemble a shape similar to a building – picture a blocky skyscraper, with no detail on the outside whatsoever, just flat surfaces. We'll learn a little about how to position objects when creating them, and we'll practice the concepts of parenting while we're at it.

Our building is going to be made of three cubes. The bottom one will be the base – thicker and shorter. The middle and top will each be thinner and taller than the last. We'll center the cubes so they all have the same X and Z positions (forward/back and right/left) but raise and lower them so that they make a sort of tower, almost like a stack of blocks. In the end, it will look something like Figure 4-3.

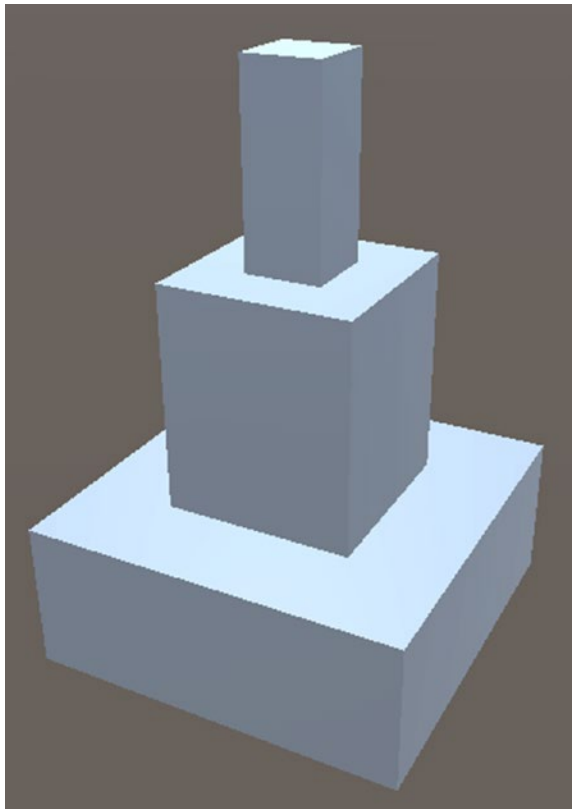


Figure 4-3. *A look at our Skyscraper GameObject*

First, before you embark upon this quest, make sure you're not lost in some awkward orientation within the Scene view. Within the Scene window, you can use the little gizmo in the top-right corner to see how your camera is rotated, almost like a compass (shown in Figure 4-4). The arrows are color-coded by axis, as we mentioned before. If you'll recall

- The **X axis** is **red**, and it corresponds to **right** and **left**. Increase the X to go right. Decrease the X to go left.
- The **Y axis** is **green**, and it corresponds to **up** and **down**. Increase the Y to go up. Decrease the Y to go down.
- The **Z axis** is **blue**, and it corresponds to **forward** and **back**. Increase the Z to go forward. Decrease the Z to go back.



Figure 4-4. *Gizmo in the top-right corner of the Scene window. The axis that the arrow corresponds to is written beside it, and each arrow is color-coded by axis*

The arrows all point along the positive direction of their axis. In other words, they point to the actual world direction of right, up, and forward. So the green arrow, corresponding to the Y axis, is always pointing up. This means if that arrow is pointing down for you, then your camera is upside-down. The gizmo is telling you where “up” really is. Realign your camera by holding right-click while moving the mouse until you have the green arrow pointing up, as it is in Figure 4-4.

Now create a cube. Make sure it's selected and change its name to Cube Base at the top of the Inspector. We'll make it thick, but short. Using the Inspector, set its Transform scale to something like (10, 4, 10). That's 10 on the X and Z axes, but only 4 on the Y axis (height).

There are several ways to ensure that one object is aligned with another when you create it. Every object you create is placed at some certain spot in front of the camera. If you create another cube without ever moving your camera since you made the first one, they'll both be in the same spot.

If you ever use the shortcut that we described earlier to center your camera on an object, which is done by pressing F, you've also positioned your camera such that any new object created will have the same position as the object you've focused on. It puts

the camera in a place where the objects you create will go exactly on top of that focused object.

So if you've moved the camera since creating your Cube Base, you can simply make sure it's selected and then press F while your mouse is over the Scene window. This will put focus on the Cube Base, moving the camera to look at it. Any new objects you create will be placed in the same position of the Cube Base, so create another cube now, and it should be in the same spot. You can also simply copy-paste the values from the Cube Base position to that of the new cube using the Inspector.

But all of these objects are meant to be attached to each other, so we know we're going to use parenting anyway. It makes sense that the upper cubes would be attached to the lower cubes, right? So when you create your new cube, name it Cube Middle and make it a child of the Cube Base. This makes it even easier to see if they're aligned correctly, since the Inspector will now be showing us the local position of Cube Middle. If they're in the same spot, Cube Middle will have a position of (0, 0, 0).

We want to make sure it's centered on the X and Z, so if the X and Z position values of Cube Middle are not 0 already, set them to 0. Now you can simply use the position and scale transform tools (W for position and R for scale) to move the Cube Middle up above the Cube Base and scale it up until it's a good size, if you want to go by eye.

Do the same thing for a third cube, named Cube Top. If you want, you can copy-paste Cube Middle (Ctrl+C, Ctrl+V) and then rename it, so that it starts right at the same position, or start fresh with a new cube and set its local position to (0, 0, 0) to center it again. Make it a child of Cube Middle, not Cube Base, because the Cube Top is technically attached to the middle cube – that is to say, if we wanted to just rotate or move the middle cube, we would expect the top cube to rotate or move with it.

You can once again scale and position the Cube Top, just as you did for the others. Leave its local X and Z positions at 0 so it's centered, and raise it so its bottom rests on the top of the cube beneath it.

Now you have the Cube Base, which is the parent of Cube Middle, which in turn is the parent of Cube Top. You can make those upper cubes look however you like. It isn't really important. If you want them to look just like our version of the Skyscraper in Figure 4-3, set their **world scale** to

- (10, 4, 10) for Cube Base
- (5, 6, 5) for Cube Middle
- (2, 5, 2) for Cube Top

Note that I say **world scale**. Like the position, the scale can also be portrayed either as local or world. The local scale is relative to the scale of the parent. The X, Y, and Z values of the child are multiplied by the corresponding X, Y, and Z values of the parent. This can make it a bit harder to work with, since you have to do some extra math to figure out exactly how many units the scale actually comes out to.

Instead of doing that math, you can just unparent the cubes so they go back to using world scale, set the scale, and then parent them again. You'll notice when you parent or unparent a cube, its scale values will change, although its size doesn't appear to be any different. Like with the position, that's just the scale going from world to local (or vice versa). They still represent the same physical size; it's just a matter of whether it's relative to the parent or not.

Pivot Points

With our cubes all put together, let's demonstrate another important concept when it comes to hierarchies of objects. A pivot point is the point around which an object will pivot when it is rotated. It is also the point on the object that will be at the exact position of the object.

Select your Cube Base and switch to the position transform tool (hotkey W). This will show the gizmo of three arrows in the Scene window. Keep pressing Z, and you'll notice that the gizmo position will change back and forth between two positions.

This is because we're toggling the tool handle position control, which is located just to the right of the transform tool buttons in the toolbar. It's a little button that either says "Center" or "Pivot," which you can click to toggle, or you can just use the Z hotkey. This depicts where the transform tool gizmos are drawn: either at the center point between the object and all its children or at the pivot point of that specific object.

So if you want to know what the pivot point is for an individual `GameObject`, just make sure that button says "Pivot" and then click the object and look where the gizmo is drawn. For a cube, the pivot point is at the center. This means if we rotate it, it rotates around its center.

The pivot point is also the point on the object that will be positioned at the exact point that the Transform position is set to. So if the cube is positioned at (5, 5, 5), then that means the center of the cube is at (5, 5, 5), not one of its sides, not its bottom or its top. This can be a pretty important detail to know when positioning objects.

Every mesh (3D model) has a pivot point. For these simple shape meshes that Unity provides us, like cubes, spheres, planes, or cylinders, it's always the center. But at some point, you might use meshes you find on the Internet or meshes designed by artists you're working with or even meshes you made yourself. If the pivot point is somewhere weird, it's usually pretty obvious once you get to using the mesh.

For example, let's say your artist provides you with a mesh to use for a gun that your player character is supposed to hold in their hand. You make a `GameObject` for the gun, and your code positions the gun at the position of the player character's hand. But the gun isn't showing up in the player character's hand. It's somewhere off to the side. This is a pivot point problem. Remember, the object's pivot point is the point of the mesh that actually goes where the object is positioned. If you're going to position the object at the player's hand, then the pivot point should be at the handle of the gun. That way, wherever we position the object, that's where the handle is – not the barrel or some random sliver of air two feet away.

Lucky for us, we aren't dealing with anything but the basic shape meshes, so everything should be pretty predictable. Still, there will be times where we'll want to adjust a pivot point for an object ourselves – and it's not hard to do this.

Let's say, in our game, we had buildings like this skyscraper we just made and we wanted to let the player purchase and place these buildings wherever they liked on the playing field. This pivot point becomes a problem for us, because we can't just use the surface position of the floor to place our cube there – we'd only be placing its center on the floor, and the rest of it would clip through the floor, sticking out beneath it and not showing on the camera. We would have to always position them upward by half the size of their base cube to get them neatly placed on top of the floor. But if each building has a different height for its base cube, this becomes a hassle to code.

We want its pivot point to be at the center on the X and Z axes, but at the very bottom on the Y axis, so that when we say “place the building at this point on the floor,” the bottom of the building is at that point. This way, no part of the building ends up sticking through the floor.

This is a simple fix. We can make an “empty `GameObject`.” This is a `GameObject` with no components except for the `Transform` – nothing but a point in space. Of course, we could add components to it afterward, but we don't need any in this case.

Create an empty `GameObject` with `GameObject ► Create Empty` or `Ctrl+Shift+N`. Name it something accurate: Skyscraper. It's the root `GameObject` of the building hierarchy – that means the master parent, the `GameObject` that holds all the

rest. When we move it, we move the whole building. So it makes sense to name it what its children make up: a skyscraper (albeit a simple one). Accurate names are a good practice to get used to!

Now we just position it where we want the pivot point to be and then simply make this empty `GameObject` be the parent of the Cube Base. I'll show you a trick to position it correctly.

Remember how I said that local positions are based on the scale of their parent? If the Skyscraper has a parent that has, say, 10 scale on the Y axis, then each point in the Skyscraper's Y position will count for 10 units. It's multiplied by the scale of the parent. We can use fractions too, so .5 would count for half the Y scale, .25 would count for a quarter of it, and so on.

We can use this to our advantage. For now, make the Skyscraper empty `GameObject` be a child of the Cube Base. Now the Skyscraper is using local positioning, meaning its position is relative to the Cube Base.

Going by what we just discussed, this means that a single point in the Skyscraper's Y axis will count for the whole height of the Cube Base. We know that if we position the Skyscraper at (0, 0, 0), it's at the exact position of the cube. Since the cube's pivot point is its center, this means the Skyscraper is placed at the center of the cube. So all we need to do is "go down" by half the height to reach the bottom of the cube. To do that, since we're using local positioning, we just give it a Y position of -0.5 . You'll recall from the preceding text that adding to the Y value goes up and subtracting from it goes down. That's why it must be negative, so make sure you put that "-" in there.

After we apply that position, the Skyscraper object should be at the exact bottom of the Cube Base, with the other axes centered on it. You can now just make the Cube Base a child of the Skyscraper object. Now the Skyscraper is the root `GameObject`, as we intended, and it's positioned at the bottom of the whole building. This makes it the pivot point. When we rotate the Skyscraper object, everything is spinning around it, so it's all pivoting at the bottom, not the center of the bottommost cube as it would before. And now if we position a Skyscraper at some point on the floor, the bottom of the building will go there, just as we would like it. We'll be demonstrating another reason for that to come in handy in the next chapter too.

Of course, you could just position the pivot point by eye with the transform tools in the Scene window and save yourself some finicking with the numbers and the parenting and unparenting. You won't get it right on the dot that way, but a little inaccuracy is unlikely to affect you in this situation. Still, it's nice to know you have things exactly as they should be.

Summary

In this chapter, we learned the following:

- GameObjects can be the **parent** of other **child** GameObjects. When the parent position, rotation, or scale changes, the children move, rotate, and scale with it as if attached to it.
- **World position** is the position of a GameObject in the scene, without any relation to other objects. **Local position** is the position of a GameObject relative to its parent. They both can be used to resemble the same position in different forms.
- The **pivot point** is the point on an object that goes exactly where the object position is set. For example, if you want to set the position of a gun to put its handle in the player's hand, you want the pivot point of the gun to be at the handle. If it's somewhere else, like the barrel of the gun, positioning the gun at the player hand position will instead place the barrel of the gun there – not the handle.
- When rotating an object, all of its children pivot around the pivot point. This can change the way an object rotates in a significant way.
- To reposition an object's pivot point, you can create a new empty GameObject, place it where you want the pivot point to be, and then make the object a child of that empty GameObject.

CHAPTER 5

Prefabs

A **prefab** is a type of **asset** you'll store in your project. They act something like a blueprint for objects. You set up a prefab for some kind of object in your game and then add instances of the prefab to your game, in any of your scenes. This connection to the prefab will remain across all the instances, and you can then change the prefab itself to automatically change every instance you've placed.

For example, you might make a prefab out of a certain enemy type. You place that kind of enemy throughout your game levels (scenes) possibly hundreds of times. Somewhere down the line, you want to make some change to this enemy type – make it a little bigger or give it a little more health or make it move a little slower. If you had simply copy-pasted your enemy `GameObject` through all your scenes, you would have to change each one individually and make sure you kept them all consistent. This can be very tedious! But since you made a prefab for the object, you can just change the prefab itself. As an asset in your project, you simply find the prefab in the Project window, edit it, and make the changes you desire. Those changes automatically reflect across all the prefab instances in all your scenes.

Making and Placing Prefabs

To create a prefab out of an existing `GameObject` in your scene, you can just drag that `GameObject` from the Hierarchy window into the Project window and drop it into a folder where you want to save the prefab asset. This creates a prefab and automatically associates that `GameObject` in the scene with that new prefab. In other words, the object you dragged and dropped is no longer some random, one-off object in this scene. It's now an instance of the prefab.

We'll create a prefab out of our Skyscraper `GameObject` we created in the last chapter. Drag the Skyscraper `GameObject` from the Hierarchy and drop it in the Project. Don't just drag one of the children, though. It must be the root `GameObject`, named Skyscraper.

The new prefab asset will be named the same thing: Skyscraper. To place further instances of the prefab – exact copies of our Skyscraper – we can just left-click to drag and drop the prefab asset from the Project window into the Scene window. As you’re dragging the prefab over the Scene, you’ll notice that Unity lets you see where it’s going to be placed, showing the new Skyscraper where your mouse is pointing. It will automatically be placed on the surface of other objects you’re pointing at, whether they be another instance of the Skyscraper or the floor we created earlier. Once you release the left-click, the instance is officially placed in the scene.

Remember what we were learning about pivot points in the previous chapter? If we had left our pivot point in the center of the Cube Base of the Skyscraper, it would not be placed so neatly on the surface of the floor when we set down prefab instances of it. It would stick through the floor. Since the pivot point is the point that will be at the position given to the object, if we want the bottom to be on the surface that we point our mouse at (and we do), we must make that the pivot point. Good thing we did that already! Now when we place the object, it lines up correctly every time.

Editing Prefabs

Place a few extra Skyscrapers onto the floor in the scene, and let’s learn to edit a prefab so we can see all our instances change when we’re done.

To edit a prefab asset, you can double-click the asset in the Project window or click it and then click the “Open Prefab” button just beneath the head of the Inspector.

This opens a sort of fake scene in Unity, where nothing exists but an instance of the prefab. Look at the Hierarchy window, and you’ll see that none of your other GameObjects are there anymore, and a bar stretches across the top of the window showing the prefab name and the little blue box icon that represents a prefab GameObject. This bar has an arrow on its left side that can be clicked to bring you out of prefab editing and back to the scene you were in before.

There’s also a similar bar stretching across the top of the Scene window, which lists, from left to right, the scene environments you’ve been in. The one you’re in now will be on the rightmost side, and the path you’ve traveled to get there will be on its left side, as shown in Figure 5-1.

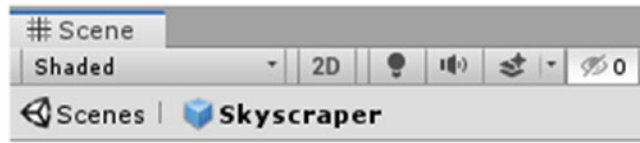


Figure 5-1. The top of the Scene view while editing our Skyscraper prefab

As of right now, it looks simple: Scenes on the left, which is the normal view where you’re looking at the loaded scene itself, and then the Skyscraper prefab, which is where we are now, viewing just the objects that are part of the prefab. This bar becomes more notably useful when you begin having prefabs within prefabs (it sounds scary, I know) and you start opening a prefab while viewing another prefab. The bar pretty much shows you the rabbit hole you’ve gone down, in proper order. At any point, you can click one of the names to switch back to that environment.

Let’s change our Skyscraper here in the editing environment. The Scene window will function as it normally does, so all our transform tools will work as they did before. Do something crazy to the Skyscraper, like grab the Cube Middle and drag it high above the Cube Base. Then, press Ctrl+S, or click the Save button all the way on the right side of the bar at the top of the Scene view. This makes the changes take effect. If your prefab has been changed and you try to navigate out of the editing environment without saving, Unity will stop you and ask if you’re sure you want to discard the changes. If you’d rather just automatically save all your changes, you can check the “Auto Save” box just to the right side of the Save button we just clicked.

Once you’ve made your change and saved the prefab, navigate out of the prefab editing environment using the top bar of the Hierarchy window (click the arrow on the left side) or the Scene window (click “Scenes” on the left side).

Now that you can see your scene again, all the Skyscrapers will have changed to reflect the prefab.

Another way to edit prefabs is to simply make some changes on an instance of the prefab in your scene. Then, to apply those changes to the prefab, just drag and drop the instance from the Hierarchy to the asset in the Project window. This makes the asset the same as your modified instance in one simple motion. For example, we could have made our change to one of the Skyscrapers in the scene we were working on and then dragged and dropped that modified Skyscraper from the Hierarchy to the asset in the Project, and all of the other instances in the scene would have updated to match it. Try it yourself, if you want. There’s no limit to how often you can change your prefabs!

Overriding Values

Sometimes, you want a prefab instance that behaves a little differently than the blueprint dictates – such as an enemy with more speed or an enemy that holds an axe instead of a hammer.

You can make little changes like this to an instance of a prefab, and Unity will keep track of what is different across your instances. These are called **overrides**. To describe it technically, when you override something on a prefab **instance**, any change to that something on the prefab **asset** will not reflect on the overridden instance.

For example, let's say we have a prefab "Soldier" representing some enemy soldier that we've placed many instances of across all of our scenes. But we want to put down an instance that has a bit more health than the rest, a particularly strong soldier. We go into the Inspector and increase the health of this soldier instance, and we rename it to Burly Soldier so we know it's special.

But later, we decide that all of our soldiers have too much health, and we want to decrease it a little. To do this, we change the prefab asset, decreasing its health a little. This causes every instance of the prefab to reflect those changes, making them have a little less health and saving us a lot of trouble finding them all and changing the health of each one individually.

However, since we overrode the health for our Burly Soldier, his health won't change. Unity will recognize that we overrode that value and leave it alone when we change the prefab asset.

This way, overrides are preserved when the prefab asset is changed, so that any one-off differences you make are not undone the next time you edit and save your prefab.

This applies to values in the components of the GameObjects associated with the prefab – including nested (child) GameObjects. This applies as well to the Transform component, so that position, rotation, and scale can be overridden for children of the prefab root (the master parent of the prefab). The root GameObject itself does not concern itself with the position and rotation of the parent, however (it would cause some pretty strange anomalies if it did). You can consider the position and rotation of the root GameObject to always be overridden – even if it is the same as that of the prefab itself, it will not be updated if you edit the prefab to rotate or position it differently. However, since the children use local position and rotation based off the root, their position and scale won't be considered "overridden by default." We'll go over this with an example in a bit.

But component values aren't the only things that can count as overrides. Removing components, adding components, and adding extra child GameObjects all count as overrides as well. So if you delete a component from some GameObject in your prefab instance and then later update the prefab asset, the deleted component will not be added back to the instance. Unity recognizes that you made a conscious override and preserves it. The same goes for adding an extra component: it won't be removed when you edit the prefab.

Component values that have been overridden will have bold (thicker) text for their name and value. The Transform of your base GameObject will have a bold position and rotation, because this is world position and isn't associated with the prefab. But the local position of children will only be bold if you have changed it from the prefab setting.

Let's override one of the Skyscraper instances we have in our scene. Using the position tool (hotkey W), just grab the Cube Middle of one of the Skyscrapers and pull it up or down a good bit, so it's noticeably off from the rest of the instances. You'll notice that its Y position in the Inspector will become bold, because it's now an overridden local position. It's not in the same place as it should be according to the prefab, so it is marked as an override.

Now that one of the instances is unique, let's edit the prefab and drag the Cube Middle back down to the position it was before (or near to it), touching the Cube Base as it should be. It's okay if it's not exact. Save the prefab and then return to the scene.

All but one Skyscraper should reflect the changes now. Since we overrode the local Y position of the Cube Middle for one of the instances, when that value is changed in the prefab, it is ignored by this one instance. We have given it a special value, and Unity will respect that and preserve it for us.

As I said before, this applies to local positions and rotations – the children of the prefab **root**. The **root** is the GameObject holding all the others, which, for this prefab, is the empty GameObject we named Skyscraper. The root GameObject is not concerned with position and rotation of the prefab. You'll notice that even if you give a prefab instance the exact same position and rotation as the prefab asset (by comparing their Transform components in the Inspector), the position and rotation of the instance is still bold. It's overridden by default. If you edit the prefab and rotate the root or move the root a few units in one direction, that change isn't going to take effect to any of the instances. This is just as well – you wouldn't really expect it to. As far as we're concerned, changes to the position and rotation of the root GameObject shouldn't affect the instances.

However, the rotation of the root will be used when placing new instances in the Scene by dragging and dropping, which can be a useful setting to specify at times.

There’s an easy way to view the overrides you have applied to a prefab instance. Select the Skyscraper instance that you have overridden the Y value for. As long as you have the root selected (the one named Skyscraper, not any of the children), you’ll see some extra options in the header of the Inspector (just under the field containing the name of the GameObject). The text “Prefab” will be on the left, and to its right side, you’ll have buttons “Open” and “Select,” as shown in Figure 5-2.

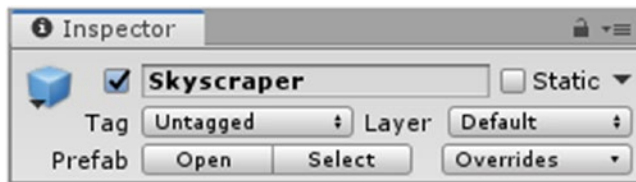


Figure 5-2. Prefab options are listed for the Skyscraper GameObject

“Open” will open the associated prefab asset for editing, and “Select” will select the prefab asset in the Project window (in large projects, this can be easier than fishing around for it yourself). To the right side of these buttons is a dropdown button titled “Overrides.” Clicking this button will show a hierarchy of GameObjects included in the prefab. Nested inside GameObjects, it will also show their components which have had values overridden, as well as components which have been added or removed and GameObjects which have been added.

The purpose of this list is essentially to show you all of the differences between a prefab instance and the actual prefab asset and to allow you to **revert** those changes, which means switch back to how it is on the asset, or **apply** them, which means make the change on the asset as well, so that it’s no longer an override – it’s just the norm. There are also the buttons “Revert All” and “Apply All” on the bottom of the list, which, as their titles suggest, revert or apply all the overrides to the prefab asset at once.

For the overrides of our Skyscraper, we should see the hierarchy showing Skyscraper, inside that (indented to the right) Cube Base, and then Cube Middle inside that; and then, inside Cube Middle, we’ll see its Transform component. Clicking the individual GameObject names will show a popup window on the left saying “No Overrides.” But clicking the Transform will cause the popup window to display two views of the Transform values side by side, as shown in Figure 5-3.

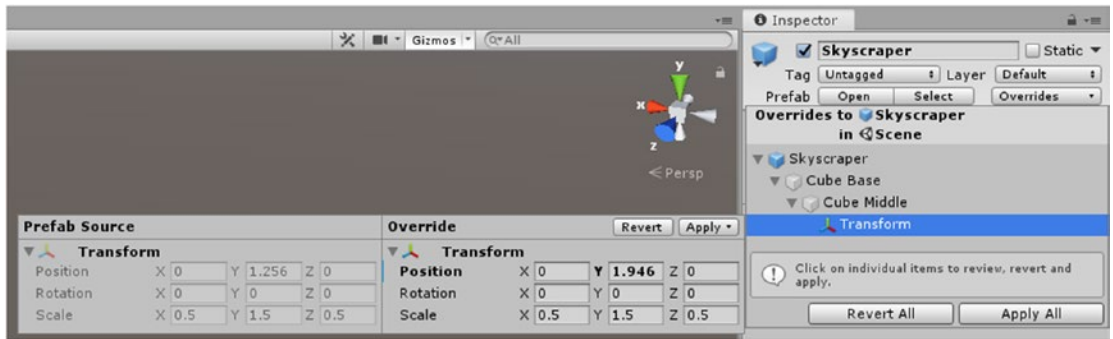


Figure 5-3. The Overrides dropdown list, with the Transform component of Cube Middle selected. This results in the popup boxes stretching out on the left side

The left side is titled “Prefab Source” and is all grayed out so we can’t edit the values. It is just showing us what the values are set to on the prefab asset and does not let us make changes. The right side is titled “Override” and shows the values of the Transform component, allowing us to edit them right there if we desire. At the top-right corner of this “Override” pane, we see the buttons to revert and apply the overridden values. This is how you revert/apply for a specific component. Let’s revert the change to make this Skyscraper go back to normal and stop defying the laws of physics. Just click the “Revert” button.

One final note on overrides: You can’t remove a GameObject from a prefab as an override. You can add them, but you can’t remove them. This is because you can’t just restructure a prefab instance like this – you must **unpack** it first, which disassociates it with the prefab altogether so that it’s just a bunch of normal GameObjects. This is done by an option available when right-clicking the GameObject. Or you can edit the prefab asset and delete the GameObjects, but of course, this applies to all instances. But neither of these is a satisfactory solution. Instead, to effectively remove a GameObject for a single instance, you can **disable** it with the checkbox on the left side of the name field in the Inspector header, which is essentially keeping the object around, but not running any of the functionality of its components: it won’t be rendered, and it won’t collide with things. This also disables all its children.

This is how you might do something like having a single instance of an enemy wield an axe instead of a hammer, as we mentioned at the start of this chapter. Disable the hammer GameObject that comes with the prefab by default, and then you can add a new axe GameObject as an override and position it where the hammer would normally be.

Nested Prefabs

One neat feature that we didn't use to have in Unity is the concept of **nested prefabs**. We can have prefab instances which are part of the hierarchy of other prefabs. Changes made to a nested prefab through its asset will automatically apply to instances of the prefab even if they are nested inside other prefabs.

For example, you might have a bunch of prefabs for different types of enemies – soldiers with male and female versions, ones with thick armor and ones with light armor, or ones which use different kinds of skills or magic. You have a variety of different weapons they'll wield – some use swords and others use spears or axes. Many of them will end up using the same weapon. So you make prefabs for the weapons and nest an instance of the weapon prefab inside each of the enemy prefabs.

This way, each enemy prefab has an instance of the weapon prefab inside it. If you ever want to change something about the weapon, you can change that weapon prefab, and all the enemies who use that weapon type will reflect the changes. If you had not used prefabs for the weapons, you'd just have separate instances of the weapon in each of the enemy prefabs, probably copy-pasting them from one to the other. Then if you wanted to make a change to a weapon, you'd have to go through each prefab that uses that weapon and change it individually.

You've probably noticed that, in the Hierarchy window, the icon to the left of the name of a GameObject is sometimes a blue cube and other times a gray cube. Perhaps you've figured it out already, but a blue cube means that a GameObject is the root of a prefab. A gray cube means it is not the root.

The color of the name also differs. Some have black text for their name and others blue. Blue text is for the GameObjects which are part of the prefab. Black text is for those which are not – they may be added as an override, but they'll still have black text because they aren't included in the prefab asset itself.

When you nest prefabs, you'll see blue cube icons as children of other blue cube icons. This helps you distinguish which pieces of a GameObject hierarchy are part of another prefab.

Aside from this, nested prefabs are pretty self-explanatory – prefabs within prefabs. But one notable difference is the applying of overrides made to nested prefabs.

The Overrides dropdown button in the Inspector will only show for the root of the prefab – not for nested prefabs. With an override applied to a nested prefab, you'll be offered two options when using the Overrides list to apply changes made to the prefab: one to apply the change as an override to the root prefab and another to apply

the change to the nested prefab itself. The difference here is whether or not the nested prefab asset has the change applied to it. Either way, the change is going to affect the root prefab, since it contains an instance of the nested asset – the question is, do you want the nested prefab, and all instances of it (nested or not) to have the change? Or do you just want the change to exist in the root prefab’s instance of the nested prefab, as an override? This just depends on the situation, but it’s an important distinction to make.

These two options will show whenever you click the Apply button when selecting an override to a nested prefab in the Overrides dropdown. Where before the Apply button would offer a dropdown list with only one option, it will now offer two: one to apply to the nested prefab itself and the other to apply as an override to the root prefab.

Prefab Variants

A prefab variant is a prefab asset that overrides another prefab called the **base**. It’s like a copy which serves as a way to make a variation of an existing prefab. Values that haven’t been overridden in the variant will be kept up to date with their settings in the prefab base.

For example, you might have variations of enemy types: a larger version of the same enemy that has more health but moves slower or a version that wields a different sort of weapon. All the base functionality is kept in the base prefab – the model, the animations, and the scripts that provide it with its AI and logic. But little customizations are made on the component values, or a weapon is disabled and a different one is swapped into its place.

This is a useful way to keep prefabs working for you while still providing you some flexibility. You can have these variations with their overrides while still benefitting from the fact that any change to the base prefab will automatically take effect on the variants, unless the variant has overridden the value. If you had instead copy-pasted the base prefab to make the variants, any change that you wish to make to some common value across the prefab and all of the variants would have to be done individually for each variant, which is the main problem that prefabs seek to solve.

To create a prefab variant, right-click a prefab asset in the Project window, select Create, and then select Prefab Variant, an option which can only be clicked if you have right-clicked a prefab. This will create a new variant next to the base prefab in the Project and will allow you to type out a name for it. Alternatively, you can create a new variant by dragging a prefab instance from the Hierarchy to the Project, which will result in a prompt asking if you’d like to create a new, original Prefab or a variant of the existing prefab.

You can even create variants whose base is another variant, if you wish.

The icon for prefab variants in the Project and Hierarchy is a blue cube, like prefabs, but it is decorated with a couple arrows to indicate that it is a variant. When you select a prefab variant, the Inspector will point at the base prefab in the header, just beneath the name of the variant.

When using variants, it's important to note that they pretty much work like a prefab instance with overrides. Anything that is different on the variant is considered an override to the base. As such, you often don't want to apply any of their overrides, because their overrides apply to the base prefab, not the variant asset. The variant asset is simply a place to store your overridden version of the prefab. You'll override what you want – change values, add components, remove components, add GameObjects, or deactivate GameObjects, whatever you need – and save the variant asset. That's all. Applying overrides is not necessary.

For example, if you were to create a “Bulky” variant of your enemy type, edit it, increase its scale, and decrease its speed and then go to the Overrides dropdown and apply all of the changes, you would end up switching the base prefab to be Bulky, and the variant would be the same as the base – which eliminates the purpose of having the variant at all. To make this distinction clear, the Apply All button in the Overrides dropdown will instead read “Apply All to Base” when working with a variant, just to give some extra warning.

Summary

In this chapter, we learned the following:

- A **prefab** is a GameObject that is saved as an **asset** by dragging it from the Hierarchy window and dropping it in the Project window.
- We place **instances** of the **prefab asset** by dragging the asset from the Project window into the Hierarchy or into the Scene.
- The prefab **asset** can be edited by double-clicking it in the Project window. Any changes made here will automatically occur on **all instances** of that prefab across all of your scenes.

- If you're creating a certain type of `GameObject` that you plan on copying and placing many instances of throughout your scenes (like an enemy or powerup), you should make a prefab for it and place instances of the prefab. You'll only have to edit the prefab once, and all of the instances will update. This can save you a lot of trouble.
- Any changes made to the values of components in a prefab **instance** are qualified as **overrides**. If the prefab **asset** makes a change to a field that the **instance** has overridden, the change will not occur on the instance. In other words, overridden values are preserved, allowing us to make one-off changes to specific instances that won't be undone the next time we edit the asset.
- A prefab variant is a prefab asset that copies from another prefab asset. A variant can be used to create a consistent different version of an existing prefab asset, such as an enemy with a different kind of weapon or with more health. Values which are not overridden are in control of the base prefab asset. Those which are overridden are in control of the variant asset.

CHAPTER 6

Programming Primer

Most people don't really know what programming is all about. They just know that "you have to be really smart to do it" and "it's complicated." This chapter will go over the basic concepts of programming to lay the foundation for beginners. If you're a more intermediate reader and you've already dabbled in writing code yourself, you can safely skip to the last topic in this chapter to learn how to prepare a script to write code in.

So what are we, as programmers, expected to do? What is programming, really? To put it simply, programming is writing code. Code is text that a computer knows how to read.

Programming Languages and Syntax

There are many different kinds of programming to serve different purposes, and one sort of code doesn't fit every situation. So we have **programming languages** for different purposes.

We're using the programming language C# (pronounced *C sharp*), because we want to code video games in Unity, and that is the programming language Unity expects and supports. If we instead wanted to code a web page like one you visit on the Internet, we would use HTML and CSS – HTML defines the layout and contents of a page, and CSS defines the appearance of a page. They're not at all like C#. Using them is a very different sort of experience, but one would still constitute it as programming.

We've taught computers how to read code written in different programming languages and to translate those languages into the effects we desire. But because the code we write is read by a computer, and computers are very technical things that cannot think for themselves, we must follow what we call a **syntax**.

Syntax is, simply put, what you write in the programming language and what the effect is. It's the set of rules that the language follows. What the computer expects to see is a very rigidly defined thing – albeit more so in some languages than others.

The computer knows only the syntax. It doesn't really know how to guess or assume what we mean to do. If we screw up the syntax even slightly, the computer will read our code, see something it doesn't expect, scratch its head, and tell us we're wrong. And we can't tell it "Come on, you know what I meant!" because it's just a computer.

Every programming language has its own syntax, its own rules for what you type and what it does. A lot of them are quite similar to each other. For example, learn C# and you can pick up Java with ease, because the languages share a lot of commonalities.

A lot of beginning programmers think very hard about what language they're going to learn. Maybe you spent a long time looking at your options before you decided on this book, because you weren't sure if you should learn C++ because it's "more respectable" or Python because it's "beginner-friendly."

But in truth, it's not such a big deal which language you start with. It's not so much about learning the right language, it's about learning how to program. Once you know how to program, learning a language pretty much boils down to learning a syntax. And syntax is just rules and keywords you memorize. It's the easy part.

So the hardest part is learning the core principles of programming. It's made harder for beginners by the fact that they're also learning the syntax of a programming language for the first time, adding these technical and at times frustrating little rules they must follow. This is why a common piece of advice for budding programmers is to *pick a language and stick with it*. Avoid wasting time with hopping from one syntax to another, learning to use this keyword instead of that one, and you free yourself to learn the fundamentals of programming.

Because the fundamental concepts are so important, we're going to spend a good chunk of time really exploring those concepts. But we're going to get our hands dirty and write code ourselves while we do it, and later in the book, we're going to start making our example projects to put the fundamentals of programming into practice.

What Code Does

So, if programming is writing code, and code is a language that computers understand, what are we telling the computer to do with our code?

What we're learning is referred to as **object-oriented programming**, or **OOP**. We call it that because it's largely to do with data, and the computer looks at that data as "objects."

Objects are data, and data can store other data inside it.

That data has some kind of type. It can be a simple type, like a number, text (which is referred to as a string), or true-or-false value (which is referred to as a Boolean or, in C#, just `bool`). Or it can be a type that we, the programmer, created ourselves to resemble some more complicated piece of data.

When we declare a type ourselves, we give it a name – a descriptive one, depicting what that type is supposed to be. Let’s say we’re making a type called `Person`.

Inside it, we might store information that relates to a person – things that every person should have associated with them, a first name, middle name, and last name or a day, month, and year of birth. These are all data that go “inside” a `Person`, often called **fields** or **members**.

In a video game, a `Person` might also have a field of data for each piece of their equipment – their gloves, boots, girdle, breastplate, and helmet. And these fields are their own type of data: an `Item` data type, which stores its own fields for name, worth, how much defense it provides, and so on.

A large part of programming is organizing your data and working with it. As programmers, we must depict how to portray, for example, a single projectile fired by an enemy in our game. We must alter its position (which is just data stored inside it) to make it move through the air. Our game engine will help us with things like collisions, but we must determine what happens if it does collide with another object, to make it deal damage if it hits the player or simply break apart if it hits a wall. And then, when a player takes damage, we must code the logic that subtracts from their health and make them die if they’ve lost all their health. All of these functionality-related things, the mechanics of a game, rely on programming to get them done. In order to program them, you must ask a lot of questions about how they work and what sort of data you will need to make available within them. Does the projectile pierce through enemies (a `bool`, true or false)? How many enemies can it pierce through at most (a number)? How fast does it move? How far can it travel before dissipating on its own?

Strong vs. Weak Typing

Our programming language of choice, C#, is what we call a **strongly typed** programming language. This means that any object we refer to will always have a type, and we cannot change that type on the fly. We declare types ourselves or use built-in types that exist by default in our programming language. To create an object, we must specify the type we want to create, like a `Person`. The declaration of a `Person` will depict what members

it stores: age, name, and so on. Each of these members has a type as well: “age” is a number, and “name” is a string.

This means that the type of an object is always known. It is clear what members it should have. If we attempt to refer to a member, like “name,” it is clear that the member exists. If we attempt to refer to a member that doesn’t exist on that type, we’ll receive an error telling us that we’re not doing something correctly. Similarly, if we try to assign the wrong type of data to a member, like assigning a number or bool to the “name” field of a Person, we’ll get an error.

In this way, strongly typed languages police you about your data. You can’t add or remove members for an object on the fly, because then it doesn’t match the type anymore. You can’t assign the wrong type of data to a member. Everything follows this structure, and if we ever stray from that structure, our code can’t run anymore.

Some programming languages do not concern themselves with data types. They have basic types for numbers, strings, Booleans, and so on. More complicated types are just “objects.” They store what’s in them, and that’s all you need to know. Want to make a Person? Just make an object and give it a first name, middle name, last name, and so on.

As you might expect, these languages are referred to as **weakly typed**. They are more flexible about how they handle data. You can assign new fields of data to existing objects on a whim. You can assign a different type of data to a field than what it normally stores and get away with it.

This can be an advantage or a disadvantage. Strongly typed languages are scarier. Many argue that they are harder for beginners to start with. They’re not as approachable. Your computer is going to yell at you (figuratively) more often about things you will probably initially think are stupid and inconsequential. But these languages always call you out. They keep you consistent, and they catch errors before your code even runs. Before the code gets a chance to be performed, the compiler can tell you that it’s incorrect, because it can see that you’re trying to do something that breaks the code – for example, assigning the wrong type to a field or referencing a member that shouldn’t exist in the given type.

These errors may otherwise have happened for a user in some niche situation you hadn’t thought about, causing your game to glitch or crash. Strongly typed languages may seem unnecessarily strict, but they’re just ensuring that everything is working as it should.

We’re not going to get into the debate of whether strongly typed programming languages are better than weakly typed ones. They aren’t better or worse, just different.

Each one has advantages over the other. If you learn other programming languages in the future, you may run into these fundamental differences. The important takeaway is that every object has a type and the members of that type cannot be changed on the fly. This makes it important for us to carefully consider how we lay out our data.

File-Type Extensions

As I said before, code is text that a computer knows how to read. There's nothing special about it. A code file is just a text file, and the only way software like Unity can know that it's meant to be a code file is because we give it the correct **file-type extension**.

This is the text coming after the period at the end of the file name. It's a way for computers to identify what a file is meant to contain. If our scripts were meant to be mere text files, we'd have the extension ".txt". That's short for text.

Programming languages all have different file-type extensions. Usually, it's the name of the language – or, if that's too large, some crumpled-up rendition of the name of the language. For C#, it's ".cs", because the "#" symbol isn't valid in a file name.

That goes at the end of our code files so that Unity recognizes them as code files and treats them as such. Write some code, save it, name it anything you want, and then add ".cs" at the end, and you have written yourself a code file. Often, the software you're working with will add the file-type extensions for you, usually without even showing you them.

Scripts

Unity refers to our code files as **scripts**. We don't use Unity to actually write the code, but Unity will detect when the code changes.

Every time we change the code and hop back into Unity, we'll see a spinning icon in the bottom-right corner of the screen. This means Unity has detected our changes in the code and it's now reading the code again to **compile** it.

Compiling, as far as we're concerned, is the computer making sense of our code and converting it to a machine-readable format. Luckily, we don't have to concern ourselves with this process.

All we really need to know is

- We change the code.
- Unity automatically notices the change.
- The icon appears in the bottom-right corner.
- The computer tries to make sense of our code.
- If it fails because we're not following the rules properly, it tells us in the Unity window known as the **Console**.

The **Console** window lists compiler errors in our code.

Whenever something isn't right, an error is thrown at us, telling us what went wrong. Sometimes they're super useful and explicitly worded. Sometimes they're really vague. It just depends on the context.

Let's start writing code. That's the fun part, after all.

Go to your Unity Project window. Right-click the Assets folder and click Create and then Folder. Name your new folder "Scripts." Right-click our new Scripts folder, then click Create again, and this time choose Script. Name the Script "MyScript". In this case, Unity automatically recognizes that it's meant to be a code file and will add the ".cs" at the end of the file without us knowing or seeing. We can tell that Unity recognizes the file as a script by the icon it displays to the left side of the file name in the Project window: the icon has "C#" on it.

Before opening the file, ensure that Visual Studio Code is your default code editor for Unity. At the top left, click the Edit dropdown menu; then click Preferences. This will pop up a Unity window showing you some options for the engine. On the left side are categories you can click, and the settings available for the selected category will show on the right side. On the left, click External Tools. The first option in the section on the right will be "External Script Editor." It will have a dropdown button you can click. Click that and look for Visual Studio Code within the options. If it is not available, you'll have to click the "Browse..." option, which will open a prompt for you to search for the executable of Visual Studio Code on your computer. This will be wherever you installed it on your computer, so navigate there and select the executable for Visual Studio Code. While you're at it, ensure that the "Editor Attaching" option is checked; then exit the Preferences window with the X button at its top-right corner.

Now you should be able to double-click the new script to open it in Visual Studio Code.

Once our code editor opens the file, we'll see what's written in every Unity script by default. This is the basic code you need to sort of "inject" your code into the Unity engine – that is to say, to write code that runs at some point during the playing of our game.

We'll go into more detail about that later. In the next chapters, we'll learn about programming with C# in general.

Summary

In this chapter, we learned the general concepts behind programming. It largely deals with organizing data and defining the functionality of the game. It's up to the programmer to implement the mechanics of a game, to make it all work.

- Code is just text that instructs a computer on what to do.
- Programming languages declare a syntax that determines what we write and what the effect is. If we fail to follow the syntax – even just a little typo or a missing symbol – we'll get errors.
- Scripts are text files containing our code. They're how we get our code into the Unity engine. We create them in the Project window and open them in our code editor.
- If our scripts have errors in them, Unity will show us these errors in the Console window. We won't be able to play the game until the errors are resolved.

CHAPTER 7

Code Blocks and Methods

We're about to go over some universal rules of code syntax to learn the structure of code files and to understand some fundamental terms and concepts.

Statements and Semicolons

The term **statement** refers to what is essentially a single instruction of code. Different sorts of statements exist to do different things.

A statement will end in a semicolon “;”. This tells the computer where one statement ends and the next is to begin. Some languages don't use semicolons at all and instead allow the end of a line to signify the end of a statement. The advantage of using semicolons instead of line breaks is that we can make one statement span multiple lines, because the computer isn't reading until it hits a line break, it's reading until it hits a semicolon. So if a statement becomes large and unwieldy, we can format it however we want – that is, we can break it into multiple lines.

Certain statements expect a **code block** to come after them. In this case, you do not write a semicolon to end the statement.

Code Blocks

In programming, code follows an almost hierarchical structure, where some code “goes in” other code. Code can go inside that code too, like boxes within boxes within boxes.

We call them **code blocks**. In C#, they are resembled by a set of curly braces: “{” to begin the block and “}” to close the block. Anything between those two symbols is considered “in” that block of code.

In the default code of our new script, you'll see several code blocks (i.e., sets of curly braces) already declared. You'll also see a bunch of words and symbols you don't understand. That's fine. It'll make sense soon enough.

As we said before, statements which have a code block coming after them don't end in a semicolon. The start and end of their code block is like their semicolon: they end when their code block ends, so to speak. When a statement expects a code block after it, we refer to it as **"prompting a block."**

Code blocks are used to create a section of code that's associated with the statement that came before it. Often, this block of code only runs in certain situations. Note that when we say that code **runs**, that means it happens, or gets performed, or "does its thing."

The easiest way to describe it, in my opinion, is by giving some examples of statements that prompt a code block and what they do with the code inside that block. We're going to learn about these in greater detail later, but for now, they'll help you understand why code blocks are so widely used and important:

- The "if" statement will evaluate a condition, checking if it is true or false. The "if" statement prompts a code block after it. That code is the code which will only run if the condition evaluates to true.
- The "else" statement can be used after the code block of an "if" statement ends. The "else" statement itself prompts a block of code. That code will only run if the "if" condition evaluated to false. In other words, if the "if" code block did not run, the "else" statement provides a code block that should run instead.
- The "while" statement is a loop. It evaluates a condition, like an "if," and prompts a block. It will evaluate the condition, and if it is true, it will run the code in the block; it then repeats, checking the condition again and running the code again if it is true, on and on until the condition is false.
- The "class" statement declares a data type (like the Person data type we were talking about before). Unlike the "if," "else," and "while" statements, the code inside a class isn't necessarily "run" on the spot. Rather, the code inside is just declaring fields (like "first name" or "last name") that the data type stores inside it. It's all just a declaration, the configuring of a template that we plan on using in different parts of our code, and the code block is used to wrap up all the code that belongs to that "class" statement.

As I said, we'll learn more about these types of statements later, and we'll write them ourselves to see how it's done. But for now, that should make it clear why we use code blocks: to create sections of code that either run only in some certain condition or that "belong to" a certain statement.

Comments

You'll notice some statements in the default script look like plain old English sentences. Notably, they are these statements:

```
// Start is called before the first frame update
[...]
// Update is called once per frame
```

These are known as **comments**. A comment is a line of code that doesn't run. It's a note that we write to document what some code does or why it does something. They're notes that the programmer writes to themselves or to other programmers who might also read their code. They're super useful, but easy to underestimate. You might think you don't need comments, but then four years later, you read the code you wrote and have no idea why you made certain decisions. Or you find that the comments you wrote are just fluffy nonsense that don't help you understand what's happening (which sucks because you have no one to blame but yourself).

Comments start with two forward slashes `("//")`. It pretty much tells the computer "Don't read this, it's not for you." We can write whatever we want there, and the computer just ignores it. If we wrote comments willy-nilly without using the forward slashes, the computer would think it was meant to be code and would throw an error.

Two forward slashes are for single-line comments. They end as soon as the line breaks.

We can also make multiline comments. They start with `"/"` and end with `"*/"`. Everything between will be a comment, even the line breaks, which lets you write a large comment across multiple lines without typing a new `"/"` for each line:

```
/*
This is a comment
that spans
multiple lines.
*/
```

Comments can also be used to temporarily disable code. For example, you can wrap a large chunk of code in a multiline comment to stop that code from running and then later come back and take out the “/*” and “*/” symbols to make the code run again. This is often referred to as “commenting out” the code. It’s useful in some situations for testing things, but it’s bad practice to leave a bunch of commented-out code lingering in your code because you think you may need it one day. If you don’t need it anymore, cut it out. If you aren’t using it right now but may need it again one day, back it up somewhere else.

Methods

For now, we’re going to skip over the first line of code and focus on the ones within the first set of curly braces:

```
// Update is called once per frame
void Update()
{
}
```

This is a **method**. Some languages would call it a **function** instead, and a lot of people use the two terms interchangeably.

Methods are a very important part of programming in C# and in many other languages too.

To put it simply, a method is a named block of code that can be referred to by name to run the code within, anywhere else in your code, any number of times. Running the code within is known as **calling** the method.

You can reuse a block of code by making it a method and then calling it all over the place. This prevents the need of copying and pasting the same code to multiple places in your project when you need to do the same chunk of code in many different places. This also means if you ever need to change the code within the method, you only need to change it in one place, rather than having to copy and paste the code in the many other places you repeated it.

It’s important to note that declaring a method is just that: declaring one, not running one. If you write a method, but never actually call it, then it might as well not exist in your code at all, because it does nothing. It’s never used.

So how is a method declared? What is this “void Update” method that Unity declares for us? Is it named “void,” or is it named “Update”?

We’ll go into much more detail on the “void” part down the road. For now, know that it’s a special keyword that you’ll learn about later.

The “Update” that comes after the “void” is the name of the method. The name is, of course, what we use to refer to the method when we want to call it. But this method is declared by default in a Unity script because the Unity engine itself will be calling the method.

As the comment says, “Update is called once per frame.” In other words, Unity will constantly call Update for us, which means the code within the method is going to be running quite often. We aren’t expected to call the method ourselves. We just let Unity do that for us.

Note If you’re interested in game programming, you might know what a *frame* is already. It’s a single calculation of the game logic run by the computer – a small bit of game time is played out each frame. Games are constantly updating: running their physics and other such game logic in a small increment and then rendering again to show those updates on the screen. These increments are the frames. You’ve likely heard the term frames per second (*FPS*) or *framerate* before. That’s the number of frames, or updates, the game gets per second. Each one will inch the game forward in time by a small increment. If the frames per second is too low, the game may look slow or choppy – this happens when the computer is taking too long to run all the operations it needs to each frame. When the FPS is sufficiently high, it looks comfortable and smooth, so we can’t tell that the whole experience is being brought to us in many tiny updates.

So let’s finally write some code ourselves to make something happen when we play our game in Unity. If you skipped the last chapter, its last few paragraphs went over how to create a script and open it in our code editor, so refer back there and come back when you’re done.

The easiest way to get a result is to call a method ourselves, one that logs a message to the Console window. Just to prove that the Update method really is being called constantly, I’m going to give you a line of code to write and explain what it means later.

In your “MyScript” file, write this statement within the code block of the Update method (in other words, between the curly braces):

```
Debug.Log("Hurray!");
```

Then, save your script with Ctrl+S or by going to File ► Save in the top-left corner of your code editor.

Navigating to Unity, we’ll see the spinning icon in the bottom-right corner, depicting that Unity is chewing on the changes we made to our code and testing the code for errors again. If you don’t see it, then it probably happened too fast for you to catch it.

There’s one more thing we need to do before we play our game to see the results. As we mentioned previously, Unity uses what we call **scripts** to get our code into the game, and scripts are components, so they can be attached to GameObjects. We wrote our code inside a script, of course – which is just a code file in our project. Now we have to add the script as a component to some GameObject in our scene.

This is essentially “creating an instance of our code.” We can add the same script to hundreds of GameObjects in our scene, and each script will have its Update method called once per frame. Each one is a separate instance with its own data associated with it – although we haven’t really declared any data for it yet.

Adding the script as a component is easy. Just left-click the script file in the Project window, and then drag and drop it onto a GameObject in the Hierarchy window or onto a GameObject in the Scene view. If you don’t have any GameObjects available, create anything, like a simple Cube or an empty GameObject, and add the script to it.

Once you select the GameObject, you’ll see in the Inspector that it now has an instance of our script listed as one of its components, shown in Figure 7-1.

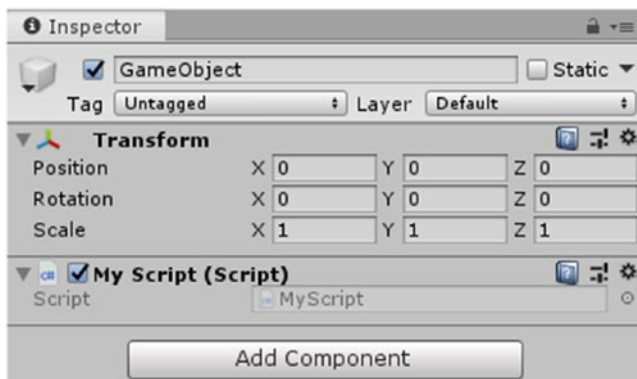


Figure 7-1. An empty GameObject with a MyScript component added

With that in place, press the Play button at the top of the screen. This will start our game (although it isn't much of a game right now). While the game is running, all the scripts we have in the scene will get updated, so we'll see the message "Hurray!" being logged many, many times over in the Console window while the game is playing. Stop playing by pressing the same button again.

Isn't that exciting? You've written your first little line of code and gotten a result out of it in your engine.

Calling Methods

Now let's examine what we're actually doing with this `Debug.Log` call we've placed in `Update`.

First off, I'll disclaim that this is simply a method call. We spoke about it before: a method is declared somewhere and assigned a name, and then other places in your code can call it by name to run the code inside it.

But what is `Debug.Log`? Is that the method name? Why's there a period in it?

The period is very important in programming. So to speak, a period "reaches into" an object, allowing you to access fields inside that object. So the method isn't named `Debug.Log`. It's named `Log`, but the method is a field inside an object called `Debug`. So we're referencing `Debug` and then "reaching into" it to reference the method `Log`.

Now you're probably wondering what `Debug` is. Maybe you don't know what "debug" even means. It's a term that programmers use a lot – it pretty much means "find and fix errors in code." To debug code is to figure out why something is going wrong and fix it so it's now going right.

But what is the `Debug` object? Where did it come from and what else can we grab from it? We haven't discussed this yet, but it's called a **class**. It's a concept we're going to get into more later. But for now, the easiest way to describe it is that the people who coded Unity wrote up this `Debug` object and gave it some useful methods and fields so that we, the consumers of the engine, can use them to help us debug our game. There are other useful features in there too, like the `Debug.DrawLine` method, which lets us draw a colored line in the world, from one point to the other. You might use that to visualize something happening in your code to confirm whether it's working how you expect.

But enough about that. We'll learn more about it later. Let's move on.

To call a method, you reference it (by typing its name, `Log`), and then you place a set of parentheses `()`. If you don't place parentheses, you're not calling the method. You're just referencing it – pointing at it, so to speak.

After the parentheses, we supply data in the form of a **string**. We spoke briefly about this in the last chapter. A string is a basic data type. It represents text. It's called a string because it's considered "a string of characters," where each character is a single symbol/letter/number.

A string is an elementary form of data that you'll use all the time in programming: names, descriptions for things like items or spells, dialog between characters in your game, and so on.

To write a string, we don't just type whatever text we want. Then the compiler would read it as code, and of course, it would throw an error when it fails to make any sense of what we wrote. We use a set of quotation marks `""` to tell the compiler that we're writing a string. Anything between those quotation marks is the text that the string represents.

You can probably guess why we write this string where we do. You saw it already – it is the text that's being logged in the Console by the `Debug.Log` method call.

This is the functionality of what we call **parameters**. Parameters are named fields of data that we "pass" to the method when we call it. When a method is declared, it can declare parameters. Each one has a name and the data type it stores. A method can have any number of parameters, or it can simply have none. Within the method declaration – that is, the code block that the method runs when called – the parameters can be referenced to get their value and make some use of it.

In this case, the parameter is a string resembling what we want to log to the Console.

The purpose of parameters is to allow a method call to be configurable, so to speak. Each call can accomplish something a little different, without requiring different method declarations with different names.

For example, let's say you had an `Enemy` script, and you gave it a method called `TakeDamage`. This method makes the enemy take damage.

But how much damage?

That can be decided by a parameter. It would be a simple number value. Whenever you call the method, you're expected to give it the parameter. You can make them take 5 damage, 20 damage, or whatever you want, just by changing the value you give as the parameter when you call the method.

This allows the same `TakeDamage` method to be called all over your code, for different results, instead of you having to make different methods that deal different amounts of damage, which would be terrible in all sorts of ways.

Basic Data Types

We just learned about how the string data type was supplied as our parameter value. Let's go over the basic data types and what the purpose of each one is, since we'll be using them quite a lot.

The major basic data types you'll be working with as a C# programmer are **int**, **float**, **bool**, and **string**.

int and **float** are number values, but there's one extremely important distinction between them: float stores a number value that can have a fractional value (a decimal point), while int does not.

That is to say, an int can be 4 or 5, but it cannot be anything between – like 4.2 or 4.68. A float can be 4 or 5 or 4.2 or 4.68 or whatever else.

string is text, as we just covered. It must be surrounded with quotes `""`. You may use single quotes `'` as well if you would rather, but it's important to note that a single symbol wrapped in single quotes, such as `'c'` or `'1'`, is considered a different data type: **char**, which is short for "character" – a single character. A string is "a string of chars," which pretty much means one or more characters arranged together – simply put, text.

bool is *true* or *false*. Those are the only two values a bool can hold. Type false or true and you've created a bool value. You might use this value type as a field representing whether or not a player is dead or an item can be sold at the shop or an unlockable ability has been learned.

Returning Values with Methods

Sometimes, you want to call a method to get some piece of data back from the call, to use it in the code which called the method. Many methods are intended to be used this way. Every method must specify a return type, which is the data type that the method will return.

The keyword `"void"` is used to specify that a method returns nothing. Call it, and you get nothing back. It's used to do something, not to return something. The `Debug.Log` method we've been using is an example of this. Its purpose is just to log a message, so it need not return anything.

Methods can return basic data types (like int, float, string, bool, etc.) or data types declared by programmers. The Unity engine has many data types declared, and many of its built-in methods return instances of these data types. For example, a method which

checks for collisions might not only return “true” if a collision was found and “false” if a collision was not found, but rather a specific data type with information about the collision: where it occurred along the tested area and information about the object it collided with. It’s data that we can then use to do whatever we’re doing – navigate around the collision or prevent an entity from moving to avoid causing a collision.

Declaring Methods

Now that we’ve learned how methods work, let’s learn how to declare our own. We’re going to start with simple methods that are pretty much useless, just to demonstrate the syntax used to declare methods – that is, what to type and why to type it.

A method declaration always starts with the return type, then followed by the method name and then a set of parentheses. As you’ve just learned, the “void” keyword specifies that a method returns nothing. Now you know the full syntax of the Start and Update methods:

```
void Start()
{
}

void Update()
{
}
```

They are really quite simple now that we’ve learned how methods work: just a basic method that doesn’t return anything.

Let’s declare a method that logs our message for us. We must write it in the same code block that the Update and Start methods are written in. Whether above, below, or between them, it doesn’t matter:

```
void LogMyMessage()
{
    Debug.Log("Hurray!");
}
```


Now, we can change our `Debug.Log` call in the `Update` method and, instead, simply call the `LogMyMessage` method and supply no parameters. Remember, although we won't be giving any parameters, we still need an empty set of parentheses `()` to signify that we are *calling* the method, not just *referencing it* (or *pointing at it*):

```
void Update()
{
    LogMyMessage();
}
```

Now if you save the script and run your game, you should notice the “Hurray!” message is logged just as you would expect. Of course, we are pretty much needlessly complicating the process by doing this, but once again, this is for demonstration purposes. We'll make proper, clever use of methods later, once we've fully grasped the concept and worked with it.

Let's learn how to use parameters. As you've learned, when we supply parameters when calling the method, they go between the parentheses. You can probably guess where we declare parameters for the method: in the parentheses after the method name, in its declaring line:

```
void LogMyMessage(/* parameters go here...*/)
```

Each parameter is declared first by typing its return type and then its name. You can declare any number of them, but each one must be separated from the next by a comma. For example, let's say we wanted to declare that “TakeDamage” method we spoke of earlier. We want a parameter for the amount of damage dealt, and let's say we also want a parameter for the “penetration” stat of the damage. The damage is a float, because we allow damage to have a fractional value. The penetration is an int, because it's a character stat and it would look a bit weird if we had some odd fractional amount for it. We would declare it like so:

```
void TakeDamage(float damage, int penetration)
```

Armed with this knowledge, you can probably figure how to declare our message-logging method with a parameter:

```
void LogMyMessage(string message)
```

We declare a parameter, using the type “string” and naming it “message.”

Note Always try to find a fitting name for your parameters. It's the message we'll be logging, so we name it "message." It could be funny to name it something like "porridge" or "heyThere," but a good name goes a long way in writing clean, readable code. In fact, proper naming can play a big part in making code that is clear enough to make writing comments for it redundant – code that, as they say, "comments itself."

So how do we use the parameter within the method code? You can probably guess – by referencing it or pointing at it, which is as simple as typing its name. Wherever we type the name of the parameter, we can imagine that the computer substitutes that name for the value stored in the parameter – which will be the string supplied whenever the method is called. We simply need to cut out that string we wrote and write the name instead. That means cut out the string quotes too – remember, if we left them, we'd just be logging the actual name of the parameter, "message," because the compiler will see the quotes and think "This is text, not a reference to a parameter."

We end up with this:

```
void LogMyMessage(string message)
{
    Debug.Log(message);
}
```

Now of course, we have produced an error. We need to give the parameter when we call the LogMyMessage method, since we cut the parameter out earlier. If you save the code now and navigate to the Unity editor, you'll see an error in the Console explaining the terrible mistake we've made – such a heinous crime, in fact, that our program falls apart and we can't even play the game anymore until we fix the error. Such is the life of a programmer.

But the fix is easy. Just change the Update method, again, and supply a string as the "message" parameter in our call to LogMyMessage:

```
void Update()
{
    LogMyMessage("Hurray for parameters!");
}
```

Operators

Wherever we are expected to supply data in our code – for example, as a parameter in a method – we can use what we call **operators** to perform equations. Operators are symbols, like a plus “+” or minus “-”, which combine two data pieces and return them in some modified state. The most obvious example would be simple math equations: adding, subtracting, multiplying, or dividing numbers.

A very important operator is “=”. Often called the **assignment operator**, it can be used to assign a new value to some named piece of data. For example, we can reassign the value of our parameter “message” to something else if we want to, by typing the name “message” and then using the “=” operator and then typing a new string value to assign:

```
void LogMyMessage(string message)
{
    message = "Something else.";

    Debug.Log(message);
}
```

This directly modifies the value of “message” so that when we log the message with `Debug.Log`, it is “Something else.” instead of the value that was given in the parameter call (essentially making the parameter useless, for demonstration purposes).

Different data types can be operated on in different ways. For example, a string can’t be divided, subtracted, or multiplied – but it can be added to another string with the “+” symbol, essentially gluing the left-side string to the right-side string, returning it as one. For example, we could prefix the message we were logging by typing out a string and then adding the value of “message” to it with a “+” operator:

```
void LogMyMessage(string message)
{
    Debug.Log("A message is being logged: " + message);
}
```

If we wanted, we could instead modify the “message” value to prefix it and then log it afterward, using a mix of both the “=” operator and the “+” operator. Take note that we are referencing the value of “message” while assigning a new value to “message” itself – of course, the old value is what we’re getting from the reference:

```
void LogMyMessage(string message)
{
    message = "A message is being logged: " + message;

    Debug.Log(message);
}
```

The operators for common mathematics are

+	Addition
-	Subtraction
*	Multiplication
/	Division

Many operators can be used in a single equation, and we can also use sets of parentheses “()” to change the order of operators occurring. As you may already know, if no parentheses are present in an equation, division and multiplication will always happen first, from left to right; afterward, addition and subtraction will occur.

As an example, this equation

$$A + B * C$$

will naturally flow like this

$$A + (B * C)$$

because the “*” and “/” operators will always occur first, in a left-to-right order, and then the “+” and “-” operators will occur after – once again, in a left-to-right order. However, we could write a set of parentheses around “A + B” to change the order of operators:

$$(A + B) * C$$

By grouping the “A + B” portion of the equation into its own set of parentheses, we have changed it so that the “*” operates second, on the result of “A + B,” instead of first, on “B” alone.

Summary

Now we’ve learned how methods work, how to declare them, how to declare and use parameters, how to return values with methods, and how values can be provided in more complicated ways using operators and sets of parentheses. Next, we’ll learn how to incorporate some logic into our code.

CHAPTER 8

Conditions

We're about to learn how to use the `bool` (true or false) data type to check conditions – in other words, to run code only if some value evaluates to true.

The “if” Block

The “if” block is a means of evaluating a condition and, if that condition is true, running the code inside the “if” block. The code is somewhat self-explanatory: we type the keyword “if,” then a set of parentheses `()` in which we type the condition to test, and then a code block `{ }` to run if the condition is true:

```
if (/* Conditions go here */)
{
    //This code only runs if the condition is true.
}
```

A condition is simply some value (often called an **expression**) which results in an instance of the type `bool` – as we've established, the `bool` type can only store true or false as its value.

Let's exhibit conditions with an example a bit more interactive than our previous ones. We're going to check for the user pressing a button, using a built-in Unity method, and log something when the user presses the button.

The method we'll be using is `Input.GetKeyDown`. This is a method which checks if the user has begun pressing down a key at just this moment. If so, it returns true. If not, it returns false. An alternate version of this method, called `Input.GetKey`, checks not if the key was just pressed down, but if the key is currently being held down. They both take a single parameter, which specifies which key we want to check. We can check any key on your average keyboard: letters, numbers, or function keys (like F1, F2, etc.), whatever we want.

To effectively use a method like this, we need to call it from the Update method. As we've said before, the Update method is constantly called. To put it more specifically, it's called once per frame – so once every time our game logic is being run by the computer to keep the game moving forward in time.

Some games limit framerate to a certain cap, like 60 frames per second (roughly .016 seconds between each frame). It won't go higher than that, even if it could. Other games do not use this tactic and simply run the code as fast as the computer can run it, which is becoming more popular. For a game like ours, with nothing rendering to the screen and no real game logic being calculated each frame, even a low-end computer should be able to run many frames per second. As soon as an update occurs after the user presses the key, the Input.GetKeyDown function will return true – right there in that next Update call. Until then, it'll be returning false, likely hundreds of times per second.

Let's give it a whirl. Write this code for your Update method:

```
void Update()
{
    if (Input.GetKeyDown("c"))
    {
        Debug.Log("C key was pressed.");
    }
}
```

Save the code and go play the changes. Press the C key on your keyboard, and you should see the message logged in the Console window. It'll be logged once each time you press the C key.

Note that there's a Collapse feature to the Console window that can be toggled with the button visible in the top-left corner of the Console window. Clicking the Collapse button will turn it on or off. While it is on, all identical messages are "collapsed" into a single entry in the Console, with a number on the right side of the entry denoting how many times that exact message has been logged. If this is on, you may think that the message is only logging once, when you first press C, but if you take a closer look, you'll notice the number is going up by 1 per press. Toggle the Collapse feature off, and you'll see all the individual entries instead.

Collapse can be useful to turn on if you happen to get in a situation where lots and lots of the same messages are being logged, flooding the Console window and making it difficult to read individual messages.

The condition coming after an “if” block is just some expression (a value, so to speak) which results in a bool type – true or false. We’re providing this in a very simple way: by calling a method which returns a bool.

It’s worth mentioning that you can shorten the “if” we wrote to make the code a little smaller if you want. Whenever you have an “if” with just one line of code inside it (one statement, ending with a semicolon), you can simply remove the curly braces. The “if” corresponds to the next statement coming after it; then it automatically ends immediately after that statement. If you ever want to add more statements to an “if” declared this way, you’ll have to go and add the curly braces:

```
if (Input.GetKeyDown("c"))
    Debug.Log("C key was pressed.");
```

If you’re doing it this way, you should always make sure to indent the statement coming after the “if,” as we have in the preceding code. It’ll still work if you don’t, but the indentation makes it visually obvious that the statement is associated with the “if.” If you use the same indentation for the “if” and its contained statement, any other programmers who must look at it will probably come after you with torches and pitchforks. I know I would.

Overloads

In C#, we can declare different “versions” of the same method, but with different parameters. These methods are called **overloads**. Their purpose is to offer varying means of calling the same method. The `Input.GetKeyDown` method has two overloads – two ways of being called. They do the same thing and have the same name but take a different type for their parameter. The first one takes a string, which should contain the character of the key to check for. The second one takes a custom data type called **KeyCode**, declared by Unity, which we’ll learn about in a second. We can supply either data type when we give our parameter, and it will work and run quite the same way.

Overloads are often used to provide more specific versions of functions containing extra parameters to further customize how the function works or versions which use a different data type or even return a different data type. A common example would be math-related methods, which often have an overload that works with the “int” data type and one that works with “float” instead. One overload will return int and/or take int parameters, while another returns float and/or takes float parameters.

Enums

The KeyCode data type is what we call an **enum**. An enum is a simple sort of object for the most part, although they have some advanced features that we won't get into just yet. They are essentially a set of names that correspond to number values. Rather than looking at these number values as meaningless numbers, we look at them by their names to make things much more readable. A common basic example of when you might use an enum is to resemble a season. We could use an int and say that 0 is spring, 1 is summer, 2 is fall, and 3 is winter. Or we can declare an enum called Season, which can be one of four values: Season.Spring, Season.Summer, Season.Fall, or Season.Winter.

Declaring such an enum is pretty simple:

```
//We start with the 'enum' keyword, then provide the name 'Season':
enum Season
{
    Spring, //Inside, we place a comma-separated list of possible values.
    Summer,
    Fall,
    Winter
}
```

The KeyCode enum has many more options than our example, but serves the same purpose: rather than assigning a number code to each key and having to refer to some chart to see which number to use for which key, we use an enum and type out the actual name of the key. Internally, the computer converts the name to a number and deals with it like that. But we get to see what we actually mean, not some meaningless number. Rather than, for example, the “A” key being 0 and “B” being 1, we simply type KeyCode.A and KeyCode.B. KeyCode contains a value for each key you might find on a keyboard, from letters to numbers to arrow keys and symbols.

This is a somewhat cleaner means of supplying a key to an input function. If the KeyCode you type compiles with no errors, then you're good. All the options you can choose are clearly defined. With a string, however, there's the danger of passing an invalid string to it. For example, we could type “page up” instead of “page up” – the only difference being that we accidentally put an extra space in the first one. If we typed the name incorrectly with the KeyCode, our code would throw an error leading us right to the problem. But if we mistype the string, either it will fail quietly and always return false

(the worst possible outcome) or it will throw an error only at runtime when the code is reached, depending upon how the method is implemented.

Let's switch our usage of the `Input.GetKeyDown` function to use `KeyCode`. Just replace the "c" string parameter with `KeyCode.C`:

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.C))
    {
        Debug.Log("C key was pressed.");
    }
}
```

Save and play, and the changes shouldn't even be noticeable. Everything should behave just as it did before.

Enums work well in many situations like this, where one might otherwise think to use some number code or a handwritten string. Usually, it's in situations where various options or settings are available and a value is expected to be one of those things. Some game-related examples would be the class or profession of a character (warrior, archer, cleric) or the type of an item (armor, weapon, consumable, material).

The "else" Block

Now that we've learned how to use an "if" block, let's take a look at a related block, the "else." You might be able to guess what it does. It's a way to provide a block of code that will run in the case where the "if" condition returns false, not true. It's code to run instead of the "if" block code.

An "else" block is declared simply by typing the "else" keyword after the closing curly brace "}" of an "if" block. Then, you write a block for the contents of the "else" block.

To demonstrate, let's change our `Input` method call from `GetKeyDown` to `GetKey`. Once again, the only difference is that `GetKey` will return true so long as the key is held, while `GetKeyDown` will return true once on the frame the key was first pressed and will not return true again until the key is released and then pressed again.

So let's log a message when C is held and a different one when it is not held – notice that we omit the curly braces for style points, as we established earlier:

```
void Update()
{
    if (Input.GetKey(KeyCode.C))
        Debug.Log("C is held");
    else
        Debug.Log("C is released");
}
```

Of course, with the rate at which Update is called (likely hundreds of times per second), you'll see lots and lots of messages appearing in your Console, all the time. That's expected. The important part is that the message changes when we hold C.

The “else if” Block

If you want to check for a different condition to run some other code if the last “if” returned false, you can use an “else if” block. It's just an “else” block, except it has its own condition attached to it as well.

We might use it to check if a different key is being pressed:

```
void Update()
{
    if (Input.GetKey(KeyCode.C))
        Debug.Log("C is held");
    else if (Input.GetKey(KeyCode.D))
        Debug.Log("D is held");
    else
        Debug.Log("C and D are both released");
}
```

In this case, we check first if C is held, and log a message if it is. If it's not held, we then check if D is held, and log a message if it is. Otherwise, if neither is held, we log a different message.

Notice that we can still use the “else” afterward. We can put as many “else if” blocks between the first “if” and the “else” as we want. And, of course, you don’t need to have an “else” at the end – you can omit it if you don’t need it.

When you chain many “else if” blocks together, remember that as soon as one of the conditions passes, any “else if” blocks below won’t have their condition checked at all. The first “if” or “else if” that evaluates to true will run its code block, and after that, all the rest in the same chain are ignored. That means if you hold down both C and D, you won’t get both “C is held” and “D is held” messages being logged at once. You’ll only get “C is held” messages, because it’s topmost in the chain of conditions – as soon as it evaluates to true, the rest of the chain is jumped over and ignored.

Operators for Conditions

As we said, the condition we’re providing after an “if” is just a value – an expression, as we call them. They’re no more than an equation that results in a bool value, true or false. It can simply be “true” or “false.” Or it can be a method that returns a bool, like `Input.GetKeyDown`, among many others.

But this also means we can use **operators** to form more complicated conditions. Remember, an **operator** is a symbol which takes some value on its left and some value on its right and operates on both values, returning some new value as a result. There are a handful of operators which return bool values – to ask if one value is greater than another or equal to another and so on. Let’s review these operators.

Equality Operators

The “==” operator (two equals symbols, not one) simply asks, “Is the value on the left equal to the value on the right?” It’s not just one “=” because that’s the assignment operator, which we used before to assign a new value to an existing object. To check equality, we use “==”.

An easy example is to compare some numbers. If the numbers are equal, “true” is returned. If the numbers are not equal, “false” is returned. A condition of “5 == 5” will always return true. Of course, that’s somewhat useless – you’re much more likely to, for example, check if your player is at maximum health: “`currentHealth == maxHealth`” would return true if they are and false if they are not.

You can compare strings too. They must be exactly equal, or “false” will be returned, even if a single letter is capitalized in one and not in the other or a single space is missing.

You can even compare bools too, which can result in some unnecessary fluffing of already-functional code. An example is that some coders will add an unnecessary “== true” after a bool value in their “if” block. We could’ve done this ourselves:

```
if (Input.GetKeyDown(KeyCode.C) == true)
```

This is redundant because we’re already getting a bool from the method call. It’s already going to make the condition pass if it returns true. The operator isn’t changing that. However, we could check if the key was **not** pressed down, essentially “inverting the condition,” by asking if it is “== false”:

```
if (Input.GetKeyDown(KeyCode.C) == false)
```

If the key is pressed, the method returns “true,” which means the condition evaluates to “true == false.” Of course, this means the operator will return “false” – the value “true” is not equal to “false.” We are left with the result of that operator, “false,” so the “if” does not pass and the code inside does not happen.

If the key was not pressed, the method returns “false,” which means the condition evaluates to “false == false.” Now, since the value “false” is indeed equal to “false,” the operator will return “true.” Since it returns “true,” the “if” passes and the code runs.

There’s also an operator which does the opposite of “==”. The “!=” operator, sometimes called the “inequality operator” (I just call it “false equals” in my head), will return false if the two given values are not equal. You can do the same thing in different ways by using one operator over the other – these two are the same:

```
if (Input.GetKeyDown(KeyCode.C) == false)
if (Input.GetKeyDown(KeyCode.C) != true)
```

In one case, we return true only if the left-hand value is equal to false. In another, we return true only if the left-hand value is not equal to true – which is just a roundabout way of doing the same thing. Both “if” blocks will only occur if the key was not pressed on that frame.

There’s yet another way to flip things around. The exclamation mark can be used before a bool value to flip it – if it’s true, it becomes false, and if it’s false, it becomes true. We can place an exclamation mark “!” before the method call to flip the value it returns.

So if you want to check that a key is not held down, you can do it either of these ways:

```
if (Input.GetKey(KeyCode.C) == false)
if (!Input.GetKey(KeyCode.C))
```

In the first case, we simply use the equality operator “==” to check if the value is false. In the second case, we insert a sneaky exclamation mark before the method call to flip the value, such that if the key was not held, the method would return false; then that value is flipped to true, and the condition passes.

Greater Than and Less Than

The “>” and “<” operators check if one value is higher or lower than another, respectively. The “>” (greater than) returns true if the value at its left is greater (higher) than the value at its right and false if it is not. The “<” (less than) returns true if the value at its left is less (lower) than the value at its right and false if it is not.

These two operators have counterparts: “>=”, greater than or equal to, and “<=”, less than or equal to. They simply do the same thing but will also return “true” if both values are equal.

Or

The “||” operator is often just called the “or” operator. Some languages even use a keyword “or” instead of the symbols. You probably never use this symbol. It’s called a vertical bar. It almost looks like a lowercase L, but it’s not. On a standard QWERTY keyboard, it’s made with the key above Enter (the backslash) while holding Shift.

You’ll use this operator quite a lot. Technically speaking, it takes a bool on its left and right sides and returns true if one or the other is true. In other words, if either one of the conditions is true, it will return true. If both are false, it will return false. To sum it up, it just means “or.” You use it to chain several different conditions together. For example, if you have a KeyCode enum stored in a parameter “key”, you might check if it’s one of several different keys:

```
if (key == KeyCode.A || key == KeyCode.B || key == KeyCode.C)
```

Now we’re mixing operators. We’re using the “==” operator to see if the “key” value is equal to a certain KeyCode value. This operator will give us a bool, so we can use it as a left-hand and right-hand value to the “||” operator, just as it expects.

If it looks confusing, you can separate the functionality with extra parentheses (although that would be somewhat redundant):

```
if ( (key == KeyCode.A) || (key == KeyCode.B) || (key == KeyCode.C) )
```

This makes it more noticeable how the “||” operators will be acting on the results of the “==” operators.

And

The “&&” operator (two ampersands) resembles “and”. It takes a bool on its left and right sides and only returns true if both of those bools are true. If either one is false, it returns false instead. This makes it useful for just throwing multiple conditions into a single “if” instead of making separate, nested “if” blocks for each one.

A simple example of its use would be to check if a number is within a minimum and maximum range:

```
if (value >= 3 && value <= 6)
```

This checks if the given value (which should be a float or int) is between 3 and 6. To read it out: If “value” is greater than or equal to 3 and “value” is less than or equal to 6.

Summary

In this chapter, we learned the following:

- How to use an “if” to run a block of code only if a condition evaluates to true.
- How to chain “else if” and “else” blocks with an “if” block. Remember, no more than one block in the chain will ever run at a time!
- An “enum” is a simple collection of names that can be used to identify a set of options instead of using number codes which give no useful context or strings which are prone to mistyping errors that the compiler won’t catch for us.
- How to use the `Input.GetKey` and `Input.GetKeyDown` methods with the built-in `KeyCode` enum to test if a key is being held down (`GetKey`) or was just pressed on this frame (`GetKeyDown`).
- How to use various operators like `+`, `-`, `||` (or), `&&` (and), `==`, and `!=`.

CHAPTER 9

Working with Objects

Remember when we talked about how programming is largely concerned with organizing and manipulating data? We interact with data as **objects**. A single piece of data is called an **object**, and an object might have other objects stored inside it – as we described in our previous examples, a data type resembling a “Person” might have a first name, last name, birthdate, and so on. We’ve already dealt with some basic objects: `bool`, `int`, `float`, and `string` are all objects as well. Now let’s learn about creating our own objects and dealing with other, less basic types of data.

Classes

A class is a type of object. Classes are how we might depict things like “Person” or “Item” as we described before. They’re the syntax we use to say “we plan on using this data type, and it stores these members” – where the members are other data inside it (sometimes also called fields). The class depicts what sorts of objects are stored inside it, what names we use to refer to them, and what type each one is. We declare classes to serve as something of a template for the creation of objects: declare a class, and elsewhere in your code, create instances of it. Each instance stores the same members but is its own unique set of those members – a copy of the base template, so to speak. So when we declare a class, we aren’t creating an object on the spot. We’re just defining a type of object.

To declare a class, you use the “`class`” keyword followed by the class name and then prompt a block:

```
class Person
{
}
```


Classes can only be declared within certain kinds of blocks. You can't declare a class within a method, for example.

Looking at our code again, you may notice that all the code we've been writing has been inside a class block this whole time – including our Update and Start methods:

```
public class MyScript : MonoBehaviour
{
    //etc.
}
```

There's some extra syntax there – the “public” keyword and the “: MonoBehaviour” bit at the end. We'll learn more about that later.

For now, let's write a simple class and play with it. We'll write a class resembling a simple item in a game. We'll declare it (i.e., write the code) inside the “MyScript” class code block we just mentioned, and we'll cut out all the code we wrote inside our Update and Start methods. You should end up with this:

```
public class MyScript : MonoBehaviour
{
    class Item
    {
    }

    // Start is called before the first frame update
    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

Now we have an Item class. We could create instances of it, but they won't store anything yet, because the code block coming after the “class Item” line is empty, so they'll just be empty objects.

Variables

When we declare members of a class, we are declaring **variables**. A **variable** is a named member which stores an object. Variables always have a type, depicting what kind of object they store. This type can be one of the basic types we’ve already used (int, bool, etc.) or any other class. Remember, classes are like a template for a certain type of object. So, for example, our Item class could store another Item inside it, as a variable.

Let’s add some variables to our Item declaration, and then we’ll dig into the syntax:

```
class Item
{
    string name = "Unnamed Item";
    int worth = 1;
    bool canBeSold = true;
}
```

Variables are declared as “[type] [name] = [value];”

In the preceding example, we’ve declared three variables:

- A string named “name” with the value “Unnamed Item”
- An int named “worth” with the value 1
- A bool named “canBeSold” with the value “true”

You can declare a variable without the “= [value]” portion – for example, we could have just written “bool canBeSold;”. If you do this, the value will be initialized to a default value. For int and float, this means 0; for strings, it’s the value **null**, which is what you get when an object is expected but one was never assigned; for bool, it’s always false.

These variables are considered **instance variables**. They exist on each instance of the class.

But we can also create variables in other code blocks. In this case, they’re called **local variables**. They’re declared the same way, but they’re not attached to any sort of object. We can just reference them by name when we want. However, they only exist within the block of code they were declared inside and, of course, below the variable declaration itself (not above it). For example, if we declare a local variable inside our Update method, that variable only exists in the Update method code block and in any blocks nested therein. This is why we call them local – they exist only in their local block, the one they were declared in.

Let's exhibit the creation of an instance of our class. To play around with this, we'll write our code in the "void Start()" method declared in our script by default. It's similar to the Update method that we talked about before. It's a method that the Unity engine will automatically call for us. Update is called once every frame, but Start is called just once when the script first initializes. For a script that is part of the scene, that means it will be called right when the game begins playing. If we were creating a GameObject on the fly, its scripts would have their Start called right as it's created, before any Update calls go through.

This gives us an easy way to run our code for testing. We'll start by writing the following code **inside our Start method**:

```
Item item = new Item();
```

This is a variable declaration: first, you'll notice the type is "Item", which is the type we declared ourselves. We name it "item". This is a common naming convention – the names of types and methods will have a capitalized first letter (like Item), while the names of variables will have a lowercase first letter, and if they're more than one word, then any words after the first will have a capital first letter to make them readable (as in "canBeSold"), which is known as camel case.

We then assign a value to the variable with "=", after which we see the "new" keyword, followed by "Item()" which is much like calling a class like it's a method. This gets us an instance of the class and is known as a constructor call.

A constructor is pretty much exactly like a method, but it's used to return an instance of a class. In this case, we're calling the default constructor – we didn't declare it ourselves. Later, we'll declare a constructor that has parameters (just like a method) for the variables in the class, which lets us assign the variables when the class is created. But let's get into that later – for now, let's demonstrate some interaction with our instance.

Accessing Class Members

After the local variable is declared and we've called the constructor with the "new Item()" syntax, we can reference the variable by its name, "item", and, as we exhibited before, use a period to "reach into" the object and access its data. Of course, in this case, the data inside it would be those variables we declared in the class block: name, worth, and canBeSold.

We can assign values to the members here by accessing them and using the “=” operator. Let’s assign some values to our new variable:

```
void Start()
{
    Item item = new Item();
    item.name = "Goblin Tooth";
    item.worth = 4;
    item.canBeSold = true;
}
```

First, we declare the local variable and create the instance; then, we have three separate statements, each one assigning a value to one of the variables inside the item. We reference those variables by using the period “.” to “reach into” the Item instance and access the data stored inside. Of course, these variables are declared in the Item class, and if we tried to access a name that isn’t declared in the class, it would result in an error preventing us from playing the game. And, as we went over earlier, since C# is a *strongly typed* language, it won’t allow us to, for example, assign a string value to “worth”, because it expects an int. It’ll throw an error in that case too.

And speaking of errors, it looks like we’ve got some waiting for us now. If you’ve written the preceding code, saved, and navigated to Unity, you’ll notice the Console window is showing three errors. This leads to the next concept of object-oriented programming that we must address: access modifiers.

An **access modifier** is a keyword you write before a nonlocal variable declaration. There are three options: **public**, **protected**, and **private**. They determine whether a member, such as the three variables we declared in the Item class, can be accessed from outside the class itself.

The **public** modifier means that the member can be freely accessed from outside the class block.

The **protected** modifier means that the member can be accessed only from within the class block or by classes which “inherit” the class – we’ll get into that concept a few chapters down the road, though, so don’t worry about it too much for now.

The **private** modifier means the member can only be accessed inside the class block itself.

We didn't provide an access modifier at all when we declared our three variables (name, worth, and canBeSold). When you don't provide an access modifier, it always defaults to private.

This is the cause of our error: we can't access the variables from outside the class block itself because they're private, but we're trying to.

The fix is simple. Put the keyword "public" before each variable declaration in the Item class, so it looks like this:

```
class Item
{
    public string name = "Unnamed Item";
    public int worth = 1;
    public bool canBeSold = true;
}
```

Save your code and the errors should go away.

Now let's use those values in a Debug.Log call – just to see something happening and make sure the values are what we expect them to be.

In our Start method, add the following line – it must be below the variable declaration and the assigning of the values:

```
Debug.Log("This " + item.name + " is worth " + item.worth + " golden coins!");
```

Here, we're chaining many "+" operators together to add referenced values together into one string. In the end, we'll expect to have a string saying "This Goblin Tooth is worth 4 golden coins!" And if we changed what we give to the "name" or "worth" variable, the message logged would change as a result.

You may have noticed we're mixing types here. The "item.name" is a string, so that makes sense – a string can be added to a string. But "item.worth" is an int, yet we're trying to add it to the string. This works just as you'd expect it to – the value of the int is added as number characters to the string, and the resulting string is returned. Some base types can quietly convert to other types like this.

So save and play – here's a quick recap of what your code should look like now:

```
void Start()
{
    Item item = new Item();
    item.name = "Goblin Tooth";
```

```

    item.worth = 4;
    item.canBeSold = true;

    Debug.Log("This " + item.name + " is worth " + item.worth + " golden
    coins!");
}

```

You should get a message in the Console window just as you would expect.

Instance Methods

Methods can go inside classes. When you do this, you’ve created an **instance method**. They’re folded up inside the class itself, so that means you must reference an instance of a class, reach into it with a “.”, and then call the method by name.

Because the method is attached to an instance of a class like this, it can seamlessly access any of the variables that belong to the class. You can type “name” to reference the “name” variable or “worth” or “canBeSold”.

Let’s move our Debug.Log call into an instance method. First, we’ll declare the method in the Item class block. Methods have access modifiers too. We haven’t used them yet, but now that we’re declaring a method inside a class, we need to make sure we designate it as “public” so we can access it from outside the class later. It returns “void” (nothing), it will be named LogInfo, and it has no parameters, so an empty set of parentheses “()” after the name:

```

class Item
{
    public string name = "Unnamed Item";
    public int worth = 1;
    public bool canBeSold = true;

    public void LogInfo()
    {
        Debug.Log("This " + name + " is worth " + worth + " golden
        coins!");
    }
}

```

Now you'll notice that we've typed the same `Debug.Log` message, except this time, we've taken out the "item." before the "name" and "worth" references. Since the method is inside the class block, it must be called through an `Item` instance, and thus, the method has access to all the members of the `Item` by default. And by "members" I don't just mean the variables – if we declared other methods inside the class, we could reference them just by their name as well, and since we're inside the class block, we could reference them even if they were private.

Of course, this won't change anything when we play until we actually call the method instead of running the same old `Debug.Log` line we had before. Replace your old `Debug.Log` line (in the `Start` method) with this:

```
item.LogInfo();
```

Save and run the game. You should see quite the same message as before. You might think, "Well, what was the point of all this, then?" After all, haven't we just written more code than we had to and accomplished the same result? Instead of the `Debug.Log` line in the `Start` method, we now have a different line, and we declared the method in the class itself!

Well, there's a little rule of programming called "**Don't Repeat Yourself**" or "DRY." The main point of this is that we now have the method and can call it whenever we need to do the same thing again. Say we wanted to log the same information for a different item, somewhere else in our code, and we never made our instance method. We just copy-paste the `Debug.Log` call over and we're done, right? But what if we want to change what the message logs? Now we have two instances of the same code to change. What if we'd copy-pasted it 20 times already? We've created a bunch of extra work for ourselves.

There could be a great benefit to having a single place for the code. By creating a method for it, we've made sure that it exists in one place only, even if it's called from many other places, and so if we ever want to change it, we need only change it once. This is one of the reasons why we say Don't Repeat Yourself. If you find yourself copy-pasting code all the time, you could probably be doing something in a more efficient and clean way, and you might be setting yourself up for heartache somewhere down the road.

As well as this, it splits the code up into relevant portions. The code which logs item-related information is kept neatly folded inside the `Item` class itself, not inside our code which implements the class. The implementing code simply reaches into "item" and calls a method – no parameters necessary.

Let's expand on this method and get a little more functionality into it. After all, you've learned enough by now to code a method that's more than just one line, haven't you?

We're going to make the method log something different based on whether the item "canBeSold". We do this with a single "if" and "else" block. Remember, since "canBeSold" is a bool, we can just type "if (canBeSold)" with no need for an "==" operator:

```
public void LogInfo()
{
    if (canBeSold)
        Debug.Log("This " + name + " can be sold for " + worth + " golden
        coins!");
    else
        Debug.Log("This " + name + " cannot be sold.");
}
```

Since the "if" and "else" are both followed by a single statement only, we don't need to write curly braces "{}" for their code blocks, as we established before. Now, the message that's logged will be different based on whether the item can be sold. If it can't be sold, there's no need to tell the user what it's worth, right?

Now, save the code and try it out. The message should be a little different now, since we changed the text in the strings. To make sure our condition is working as we expect, you can also change the "true" to "false" in the Start method when we set "item.canBeSold" and then run the code again. The message should change as expected.

Declaring Constructors

We discussed what constructors are earlier. They're much like methods, but they're called to generate an instance of a class. When the instance is created, the code in the declared constructor is run on that instance before the created instance gets returned. When calling a constructor (making an instance of a class), you pretty much call the class by its name, as if it were a method, and you must have the "new" keyword come before it.

It's a good practice to use constructors to set up new instances of your class. Typically, a constructor will have a parameter for each variable in the class you expect to be set with each instance – whatever custom fields might require a different value each time the class is created. In our case, the three variables we declared ought to be parameters in a constructor – it's silly to use three separate statements to provide the

values of variables that we’re going to set every time anyway, right? Not only that, but constructors make it obvious to you, and anyone else using your code, which values are meant to be assigned when an instance of a class is created. They are orderly, and they set a standard for the usage of a class.

Constructors are declared inside the class block itself. They are as simple as “[access modifier] [class name]([parameters]) {...}”. The access modifier we desire is “public”. Private constructors can only be used from within the class itself – there are some cases where this is handy, but this is not one of them, so we need to make sure we type “public” because if we don’t, “private” will be defaulted.

This is how our constructor will look – written within the code block of “class Item”:

```
public Item(string name, int worth, bool canBeSold)
{
    this.name = name;
    this.worth = worth;
    this.canBeSold = canBeSold;
}
```

Before we get into the statements within, let’s review the declaration itself. Start with the access modifier “public” and then the name of the class, “Item”. Think of it like declaring a method that doesn’t have a name, just the access modifier and then the return type – after all, constructors always return the type of the class itself, since they’re used to create an instance of it. Then we do the same set of parentheses “()” we’re used to, with parameter declarations just as we declared for our methods.

Now, what are these three statements beginning with “this.”? It’s simple – they’re assigning the values of the parameters to the values of the variables in the class instance.

When we made instance methods just a little bit ago, we exhibited that you can simply type the name of a class variable to reference that variable. This is the same within a constructor. The class instance already exists, and the code is running on it immediately after. So within this constructor, if we type “name”, we get the value of “name” for the class instance that’s being created by the constructor. All the variables for this instance already exist.

But since the parameters are named the exact same thing as the variables themselves, “name” also refers to the parameter. We can’t just type “name = name;” and expect the computer to know what we’re talking about. That creates ambiguity that the computer cannot solve on its own – the compiler doesn’t make guesses like this. It needs

us to clear the situation up. So to avoid this confusion, we use “this”, which is a keyword that always refers to the instance that the code is running for – the class instance being created. By referencing “this” first, we take the ambiguity out of the situation – the computer no longer sees it and says “Wait, what?” It sees that we mean to say “set the value of the class instance variable ‘name’ to the value of the parameter ‘name’”.

The “this” keyword can be used in instance methods as well as constructors, if you ever need a means of referencing the instance itself. A very common use case for it is to avoid name conflicts as we’ve just demonstrated.

Another solution to these name conflicts would be to simply not name the parameters the exact same thing as the class variables. For example, we could put an underscore before each parameter name and take the “this.” out:

```
public Item(string _name, int _worth, bool _canBeSold)
{
    name = _name;
    worth = _worth;
    canBeSold = _canBeSold;
}
```

But this is frowned upon. The former way (using “this.”) is the norm, because it’s clear, concise, and obvious. The parameters are being applied directly to the variables themselves, so why name them anything else?

Now that you’ve added the constructor to the Item class, it should look like this:

```
class Item
{
    public string name = "Unnamed Item";
    public int worth = 1;
    public bool canBeSold = true;

    public Item(string name, int worth, bool canBeSold)
    {
        this.name = name;
        this.worth = worth;
        this.canBeSold = canBeSold;
    }
}
```

```

    public void LogInfo()
    {
        Debug.Log("This " + name + " is worth " + worth + " golden
        coins!");
    }
}

```

Using the Constructor

Now that we've declared our constructor, saving the code and checking Unity will result in an error in the Console window. A class with no constructors declared will have a single, default constructor automatically provided, which takes no parameters and returns an instance of the class with all its variables at their default settings. But once you've declared your own constructor, this default constructor will no longer exist for the class. There is now only one way to declare the class, and it takes three parameters, but we're still calling the constructor with no parameters at all in our Start method.

So navigate back to this code in your Start method:

```

Item item = new Item();
item.name = "Goblin Tooth";
item.worth = 4;
item.canBeSold = true;

```

This is where our error occurs. We're trying to make an Item without giving any parameters to the constructor. You can probably guess how we'll go about changing this to call the constructor instead. Replace the code with this:

```

Item item = new Item("Goblin Tooth", 4, true);

```

We've turned our repetitive four lines of code into one clean line. The parameters are provided in the constructor call, just like with our methods, and of course, we must follow the same order that the parameters were declared in the constructor: "name" first, then "worth", and then "canBeSold".

All the constructor code will be executed before the next line of code after we declare our "item" variable, so we know all the fields are assigned before the instance is returned to us. Everything is set up and ready for the instance to be used in a neat and consistent way.

Now if we create Item instances in a hundred different places in our game code, we can just change one block of code – the constructor declaration – if we ever need to change how items are set up when they are created.

Static Members

One last thing we’ll cover about classes is the idea of static members. They’re pretty much the opposite of the instanced members we’ve been working with in this chapter.

Instanced variables, like the variables we declared for our Item class, exist as separate pieces of data on each Item instance we create.

An instanced method, like the LogInfo method, operates through an instance of the Item class. You must reach into an Item instance to call the LogInfo method. Since it’s running through an instance of the Item, it can work with the instanced variables, as our method does to log the name and worth of the item.

Static variables, however, exist as one instance for the entire class. From outside the class, you must reach into the name of the class itself, like “Item”, to access its static members.

Static methods work in the same way, and since they can be called simply by reaching into the class name, this means the method cannot access instanced members since there is no specific instance tied to the call. For example, if we made our Item.LogInfo method into a static method instead, it would throw compiler errors when we try to access those instanced variables “name” and “worth”.

Let’s demonstrate. An easy example would be to count how many instances of the Item class are created. We’ll update the Item class to add the static members, marked in bold:

```
class Item
{
    public static int NumberOfInstances = 0;

    public string name = "Unnamed Item";
    public int worth = 1;
    public bool canBeSold = true;
```

```

public Item(string name, int worth, bool canBeSold)
{
    NumberOfInstances += 1;

    this.name = name;
    this.worth = worth;
    this.canBeSold = canBeSold;
}

public void LogInfo()
{
    Debug.Log("This " + name + " is worth " + worth + " golden
    coins!");
}

public static void LogInstanceCount()
{
    Debug.Log("Number of Item instances is: " + NumberOfInstances);
}
}

```

First, we declare a static variable, `NumberOfInstances`. It's declared just like a normal variable, but after the "public" keyword, we specify a "static" keyword as well. We start it with a default value of 0. In the constructor, we use the "+=" operator to add 1 to the `NumberOfInstances` every time the constructor is called – in other words, every time a new instance of `Item` is created.

We then declare a static method that logs the number of `Item` instances that exist, accessing the `NumberOfInstances` variable. It's made static just like the variable, by typing "public static" instead of just "public".

Within that method, any attempt at accessing an instanced member will result in compiler errors. We can't call `LogInfo` or use our `name`, `worth`, or `canBeSold` variables because the static method isn't tied to any particular instance of `Item`, and those variables exist on every instance. The `NumberOfInstances` variable, however, is tied to the class itself, so we can access it.

Let's update our `Start` method to demonstrate calling the method. To demonstrate the count going up after we create our first `Item`, we'll call it once before the item is created and once again after:

```

void Start()
{
    Item.LogInstanceCount();

    Item item = new Item("Goblin Tooth",4,true);

    Item.LogInstanceCount();
}

```

As you can see, we’re calling the method through a reference to the actual type name itself, “Item”, with an uppercase “I”. We aren’t referencing the instance we’re storing in our local variable, which is “item” with a lowercase “i”.

If you test this updated code, you should see two messages:

Number of Item instances is: 0
Number of Item instances is: 1

This demonstrates the `NumberOfInstances` going up when we create the new `Item`.

Now that you understand this distinction, you might realize we’ve already called a static method before: `Debug.Log`. “Debug” is just a class, and “Log” is a public static method declared inside it. Thus, we can access `Debug.Log` whenever we want.

This is how many of the built-in methods are exposed to us, including the `Input.GetKey` and `Input.GetKeyDown` methods we called in the previous chapter.

Summary

In this chapter, we learned that a class provides a template for a type of object. We add members like variables and methods to the class definition. Every instance of the class will have those members. Instances of classes are made by calling a constructor using the “new” keyword. We can declare our own constructors for classes to provide a consistent means of initializing new instances and giving values for their variables.

We also learned the distinction between instanced variables (those within a class) and local variables, which are declared in the body of a method (among other things). Local variables are created on the fly during the method call and cannot be accessed from outside the method.

Another important thing to remember is that methods declared within a class are instanced, meaning you must call the method through a reference to an instance of the class (like with `item.LogInfo()`). Within that method, you can access other members of the class, like its variables, directly by name. If you instead mark the method with the `“static”` keyword, it is accessed through the class name directly, not an instance.

CHAPTER 10

Working with Scripts

Now that you’ve learned about the basics of objects, let’s learn about scripts. I should warn you – we’re getting dangerously close to coding our first game.

Scripts are components for our code files. They get our code into the game. We’ve worked with a script already (MyScript) just to get our code running, but we haven’t explored them thoroughly yet.

When it comes down to it, they’re objects. A script is a code file that declares a class. The class acts as a component, allowing us to attach it to GameObjects just as we might attach cameras, lights, colliders, mesh renderers, and whatever else.

But rather than creating instances of scripts with constructors and the “new” keyword as we did with our own class, we use the Unity editor to create instances by adding script components to GameObjects. If we want to create instances of the script through code, we use built-in Unity methods for that too. Since they’re components, they must be attached to some GameObject in order to exist.

As we’ve already demonstrated, we can declare event methods like Update and Start in our scripts, giving us a means of running code at certain queues in-game. There are other events as well – some you might never use and some we’ll be using in our example projects.

Scripts are meant to serve as a piece of functionality that can be added to a GameObject. It can be as complex as all your player logic or as simple as a script that constantly rotates an object.

Often, that little piece of functionality requires variables to work. Just like with a class, we can declare instance variables for our scripts, so that every instance of the script has this data “inside it.” But with scripts, since they act as components, their variables can be exposed to us in the Inspector, allowing us to edit variable values on a per-instance basis. A script that constantly rotates an object might have a vector variable (an X, Y, and Z value) that depicts how much it rotates per second. With this exposed in the Inspector, we can use the same script to achieve different things across different GameObjects. Some could rotate faster than others, or some could rotate on one axis instead of the other.

Not only that, but we can even change the values of script variables through the Inspector while playing our game. When we quit playing, the values are reverted to their original setting. This makes it easy to test different settings for variables – for example, physics-related settings like how high a player jumps or how fast they fall – without losing your old settings.

Let’s make the simple script we mentioned that rotates a `GameObject` by some constant amount. It doesn’t take many lines of code, and it’ll give us a taste for working with scripts.

Create a script in your Project window with the Create button at the top left of the window, shown in Figure 10-1.

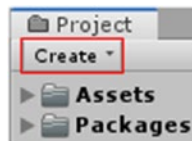


Figure 10-1. The “Create” button at the top-left corner of the Project window, just underneath the window tab

Name it `SimpleRotation`. Open the script, and you’ll be met with the base script code, as we’ve already seen once before. Let’s go over it top to bottom this time, now that you’re getting savvy with code.

Usings and Namespaces

This should be the first thing you see at the top of your script file:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

These are **usings**. They tell the compiler what other code we’re going to use in this file. We start them with the “using” keyword, followed by a reference to what is known as a **namespace**.

A **namespace** is a simple, named block of code that contains other definitions inside it. In a sense, they’re like folders for your computer. A folder can store files inside it, as well as other folders. A namespace can store definitions inside it, such as classes, as well as other namespaces.

The purpose of a namespace is to separate a relevant chunk of code from other files that won't be using it.

You can reach into a namespace to reference other namespaces inside it, as well as other classes inside it. It's much like grabbing members out of an object. You just use the period. If we declare something inside a namespace, we must access it through the namespace, typing out the namespace name, then a period, and then the definition (e.g., a class) that we want to reference. But this can be tedious. That's why we have usings.

When we include a using statement in our file, we're pointing to a namespace and saying "Give me that one." This lets us directly reference classes inside the namespace without "reaching into" the namespace to get them.

So a using statement will always point at a namespace and lets us directly reference all the definitions declared inside that namespace. They're for convenience. They shorten how much stuff we have to type.

The namespaces we're pointing to are part of the base frameworks provided to us by default. `System.Collections` and `System.Collections.Generic` are namespaces holding classes that can be used to store "collections" of other objects. We aren't going to actually use them, but they're included by default in a new script because they're commonly used. `UnityEngine` is a namespace containing most of the core definitions you'll be interacting with when coding your game in Unity, so of course, it only makes sense that it would be included by default as well.

`UnityEngine` includes things like the classes that define major components. You've already heard about some of these: `Transform`, `Camera`, `Light`, `Mesh Renderer`, and `Mesh Filter` are some examples. Of course, there's also a class for a `GameObject`.

So to sum it up, if we did not have the "using `UnityEngine`;" line at the start of our script, we would have to type `UnityEngine.GameObject` or `UnityEngine.Transform` (and so on) to refer to these classes. But since we have the using, we can just type `GameObject` or `Transform`.

While we aren't actually going to use anything in the other two namespaces, it's not going to hurt anything to leave the usings be.

Script Class

Moving on down the file, we have the class definition:

```
public class SimpleRotation : MonoBehaviour
```

The class is automatically named the same thing as our script file. This is important. You actually won't be able to attach the script to a `GameObject` as a component if you don't name the file and the class the same thing!

This extra bit at the end is what we call class inheritance. We'll learn how that works in the next chapter, but just know that the “ : `MonoBehaviour`” part is what makes the class a script that can be added as a component, not just a normal class.

The code block inside the class will be familiar from our experience with `MyScript`. We have some comments. If you recall, those are the lines that start with “//”, which are ignored by the compiler and serve as little notes for us to read. We also have a “`void Start()`” method and a “`void Update()`” method, each with an empty code block after. We learned about these before: `Update` is called once per frame, while `Start` is called just once at the start of the game.

Let's start with declaring variables for our script. We're going to use a new type of data that we haven't worked with yet. It's called `Vector3`. It's a single object that stores three floats: `X`, `Y`, and `Z`. We've seen this before. The position, rotation, and scale of our objects are all represented as instances of `Vector3` because they all have an `X`, `Y`, and `Z` value. This is just the first time we're dealing with the data type in our code.

We're going to use a `Vector3` to represent how much we want to rotate our `Transform` per second, on each axis. As you may recall, each of the three axes (`X`, `Y`, and `Z`) will tilt the object in different ways.

Declaring variables for a script is always done inside the script class (`SimpleRotation`) code block. You can put them anywhere, but pretty much everyone agrees that your variables should be kept at the top, above all your methods. When a coder (including yourself) goes to look for a variable declaration, they're going to look for it up at the top, not mixed in arbitrarily with the methods.

You're an expert now, so you should know how to declare a variable yourself. We'll make this one public, of type `Vector3`, named “`rotationPerSecond`”.

While we're at it, let's remove the `Start` method. We won't be using it, so we don't need to declare it. But we will be using `Update` in a bit, so don't remove it:

```
public Vector3 rotationPerSecond;
```

It's important that the variable is public. Protected and private variables are not visible in the Inspector, so we won't be able to customize the value for each individual script if we don't make the variable public.

Now save your code. In your Unity editor, find some `GameObject` you want to rotate. It can be one of the Skyscrapers we made earlier, or you can just add a cube to the scene. Add the `SimpleRotation` script to the `GameObject`. In case you’ve forgotten, you can do this by dragging and dropping the script file from the Project to the desired `GameObject` in the Hierarchy. You can also select the `GameObject`, go to the Inspector, and click the “Add Component” button displayed beneath all the `GameObject`’s components. This will pull down a little menu you can use to navigate to your `SimpleRotation` script, or simply type the script name into the search bar to find it and click it.

Once the script is added, you’ll see our variable listed inside it as an editable field. As you can see in Figure 10-2, it shows the name, all prettied up with extra spaces and proper capitalization so it reads “Rotation Per Second”. Beneath, it shows three number fields, one for each axis.

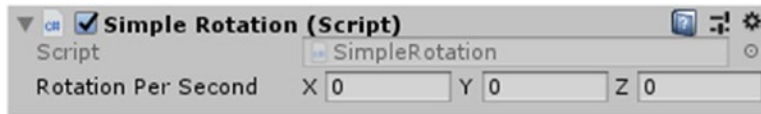


Figure 10-2. An instance of the `SimpleRotation` script component showing in the Inspector

If you don’t see the field, make sure you declared the variable as public, and make sure it’s inside the script class code block (the code block after “`class SimpleRotation`”). And, of course, make sure you saved the script file in your code editor since you added the variable declaration!

Rotating a Transform

Now, having the variable is great, but we still need to use it to rotate our object. Good news – we only need one line of code to do this, placed in our `Update` method. With the “`rotationPerSecond`” variable we declared earlier, we’ll add this line of code to our `Update` method, making the script look like this now:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```

public class SimpleRotation : MonoBehaviour
{
    public Vector3 rotationPerSecond;

    // Update is called once per frame
    void Update()
    {
        transform.Rotate(rotationPerSecond * Time.deltaTime);
    }
}

```

Make sure you save your changes in the code editor, and then go back to Unity. Remember, we don't initialize the "rotationPerSecond" to any default value, so it's automatically going to be (0, 0, 0) on all new instances of the script. This means it won't rotate at all. Select the GameObject you added the script to earlier and make sure you set its rotationPerSecond to something that's not (0, 0, 0) in the Inspector. You can use negative or positive values. Set it to whatever you want. A value of 360 will be one full circle per second, to give you a frame of reference.

Now play the game, and you'll see your rotation in effect. Congratulations. You did that.

You can also play with the value while the game is running, as we mentioned earlier. Go ahead and change the rotationPerSecond while the game is playing. Raise or lower it on any of the axes, and it should respond immediately. Once you stop playing, the value will go back to what it was when you started.

Now, let's discuss exactly what this line of code in our Update method is doing:

```
transform.Rotate(rotationPerSecond * Time.deltaTime);
```

You'll recall that the Transform component of a GameObject is what handles its position, rotation, and scale, among other things. It should only make sense that we must go through the Transform to rotate the object, right?

And since our class is a script, we automatically get access to certain members that all scripts have. One of those is "transform" – note the first letter is lowercase. "Transform" is the class type, and "transform" is our member. It's a reference to the Transform component of the GameObject that the script is attached to.

That little bit in the class declaration pointing to "MonoBehaviour" is what technically gives us access to this member – it's the part that makes it a script, not just

any old class. It's known as inheritance. We'll be learning more about it in the next chapter. For now, just know that our script class has automatic access to these useful little members that all scripts have. Another such member is "gameObject", a reference to the GameObject the script is attached to.

So we use "transform" to refer to the Transform component. Luckily, Unity provides us with this convenient "Rotate" instance method in the Transform class to rotate it. We reach in with a "." and reference that method. The method has several overloads, but the one we're using takes just one parameter of type Vector3. It rotates the Transform by the values of the given Vector3. Since rotation is an X, Y, and Z value, it only makes sense that it expects a Vector3, not a single float.

Frames and Seconds

You'll notice we're doing a little bit of math on the rotationPerSecond when we supply it as a parameter to the Rotate call. We have a "*" operator, which is for multiplication, followed by "Time.deltaTime", which is a float. You can probably guess that multiplying a Vector3 by a float is going to multiply each axis of the vector by the float. So what we get is (x * Time.deltaTime, y * Time.deltaTime, z * Time.deltaTime). But what is this Time.deltaTime float supposed to resemble?

Remember that Update happens once per frame. And your game is running at some odd number of frames per second that could change on a dime. It's probably happening hundreds of times per second, since we barely have anything going on in our scene that would slow down your computer. But during the course of playing a game, the framerate might raise and lower based on the situation. It's not reliable.

We named our variable "rotationPerSecond" for a reason. It's not rotation per frame. If we just pass rotationPerSecond for the parameter as is, we'll rotate a whole lot more than we want to. We'll be rotating by the amount we want per second, but we'll be doing it hundreds of times per second (you should try it, it looks humorous). To transform it from "per frame" to "per second" is actually somewhat simple.

"Time" is a UnityEngine built-in container for useful members like "deltaTime", which is a static float that's constantly updated and simply stores the time, in seconds, since the last frame occurred. This is often a tiny little fraction. If we're running at 100 frames per second, it'll be a mere .01. We use this as a multiplier for our rotationPerSecond to go from "per frame" to "per second."

Think about it. Let's simplify it and say we're running at a measly two frames per second. That means half a second per frame, right? So `Time.deltaTime` will be at .5 on each `Update` call. Multiply something by .5 and you get half. This means we get half the `rotationPerSecond` per call. If our rotation is 50 per second on all axes, we get 25 per frame. This is 50 in two frames. At two frames per second, that's rotating at 50 per second. This means we did just what we were looking to do.

This concept is used when we move objects as well, to move them by units per second, not per frame. It's as simple as multiplying by `Time.deltaTime`.

Attributes

An attribute is something like an instance of a class that gets attached to code definitions. It sounds odd, but it's somewhat like a way for programmers to introduce metadata into their code. A programmer might declare attributes to specify certain things about a definition. We then attach attribute instances to definitions we declare, like classes, variables, methods, and so on. Other code will then read these attributes and do something with them.

A good example can be found in some of the attributes that the `UnityEngine` provides us. For example, there is an attribute called `HideInInspector`, which we can attach to a variable to – you guessed it – hide it from the Inspector. You can use this to hide a variable while keeping it public so that other scripts can access the variable through a reference.

To apply an attribute to a definition, you'll type the attribute name in a set of square braces “[]” before the definition.

For example purposes, let's hide our `rotationPerSecond` member by adding the attribute:

```
[HideInInspector] public Vector3 rotationPerSecond;
```

The definition is the same, but we have the square braces [] and the attribute name declared before the variable. Save the code and head back to Unity. Select your `GameObject` with the `SimpleRotation` script on it, and you'll notice the `rotationPerSecond` has disappeared in the Inspector.

You can write multiple attributes for a single definition. Each one must be surrounded by its own set of square braces “[]”. You can also put a line break between

attributes and the definition, if you want to spread the definition out so it's a little more readable:

```
[HideInInspector]
public Vector3 rotationPerSecond;
```

If you had multiple attributes, you could even devote an entire line to each attribute, if you wanted.

But we don't want this attribute on our `rotationPerSecond` member, since we need to edit it in the Inspector. That was just for demonstration.

Let's demonstrate another attribute that can come in handy: `Header`. This is an attribute that you attach to a variable to cause a bold header with the text of your choice to show above it in the Inspector. This can be useful for visually separating different groups of variables to tidy up your Inspector. For example, a script representing your player character might have separate headers for variables relating to movement, jumping, and attacking.

This header takes a parameter, which is a string for the header text. Declaring attributes with parameters looks much like a method call or a constructor call. Instead of just typing the name of the attribute, we add a set of parentheses “()” with the string value inside it:

```
[Header("My Variables")]
public Vector3 rotationPerSecond;
```

Save and check the Inspector again, and you should see your bold title “My Variables” above the `rotationPerSecond` variable, as shown in Figure 10-3.

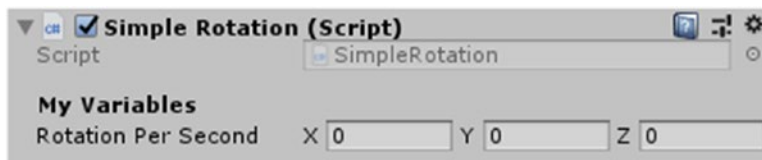


Figure 10-3. *SimpleRotation* script instance shown in the Inspector, with our “My Variables” Header attribute displaying as a bold title above the variable field

If you put the `Header` attribute on a variable that's not showing in the Inspector, such as a variable with the “[HideInInspector]” attribute or a private variable, you won't see the header at all.

These two simple attributes can be useful in keeping the Inspector for your scripts tidy.

Summary

In this chapter, we learned how to attach our code files to GameObjects as script components.

- Public variables declared in a script class can have their value viewed and edited in the Inspector. You can edit the values while playing the game to test out different settings, but those changes won't stick around after you stop playing the game.
- The class declared in the script file must be named the same thing as the script file. If the names don't match, you won't be able to attach the script as a component to a GameObject.
- Scripts naturally have access to a "transform" member pointing to the Transform of the GameObject the script is attached to.
- Attributes are declared with a set of square braces "[]". Unity declares some built-in attributes that we can apply to variables for things like hiding a variable from the Inspector or applying a bold text header above a variable for organization purposes.

CHAPTER 11

Inheritance

A major staple of object-oriented programming is the concept of inheritance.

Inheritance is when types of data (such as classes) adopt fields – such as variables and methods – from another data type.

Say you have two classes that both share nearly all the same fields, but one has an extra field or an extra method that the other does not have. Coding each one separately is tedious. They behave in the same way for the most part. If you want to change something that both classes have, you need to make the change twice and make sure it's consistent. It becomes even more of a problem if you have many classes that share most of their functionality.

Inheritance is the solution to this. You can create a **base class** holding the functionality which both classes are meant to share, all in one place. Then, other classes **inherit** from the base class to automatically share its variables and methods without having to rewrite them. Everything stays declared in one place, keeping the functionality consistent across all the inheriting classes.

Inheritance in Action: RPG Items

A classic example of inheritance is the concept of items in a roleplaying game (RPG).

Every item has certain fields:

- An int for its **worth**, which is how many golden coins it is worth in the shop
- A bool depicting whether it **canBeSold** at a shop
- A string for the item **name**
- A string for the item **description**
- An int for the item **weight**

But then we have more specific types of items. For now, let's say we have armor that goes in certain equipment slots and weapons that go in other equipment slots.

To implement this, we use inheritance. A **base class** stores the members that all items have. It resembles any sort of item, so we just call it `Item`. Now we can create classes that inherit from `Item` but have more particular uses.

Let's say every piece of equipment, whether it's a piece of armor or a weapon, has a durability value that wears down as the equipment is used. Armor loses durability as you take damage while wearing it, and weapons lose durability as you whack enemies with them.

We create an inheriting class called `Equipment`. Since it inherits `Item`, it has all those fields we give to the `Item` class, like weight and worth and so on, but we also give it an int for current durability and another for maximum durability. The current durability will go down by a point every so many hits a piece of armor takes or a weapon gives. The equipment breaks when the current durability hits 0. Repairing it will raise the current durability back to the maximum durability so we can start wearing it down again.

Now we make a class `Weapon` which inherits `Equipment`, adding fields like minimum and maximum damage, how fast the weapon attacks, what kind of weapon it is (an enum with options like axe, sword, hammer, knife, a particularly sharp stone, and whatever else), and perhaps a bool for whether it deals sharp damage or blunt damage, if we want to make some mechanical use of that in our game.

We have a separate class named `Armor`, which inherits from `Equipment` as well. We add a field for how much defense the armor provides. We'll also need an enum for the type – boots, girdle, gloves, chest, or helmet – which we use to dictate which equipment slot the armor is allowed in.

If we wanted to also have a `Consumable` class for items that can be consumed, like drinking a potion or eating some food, we could have another class that inherits directly from `Item`. Ultimately, we end up with a hierarchy of classes looking like this, where classes that are indented further to the right are inheriting from the upper class:

```
Item
  Equipment
    Armor
    Weapon
  Consumable
```

Let's get the terminology down. **Lower types**, or **subclasses**, are **more specific** than their **upper type**, or **superclass**. Item is the superclass, and more specific versions of it, like Consumable or Equipment, are lower types of it. Yet more specific versions of Equipment are Weapon and Armor. This creates a hierarchy of classes, becoming more and more particular as we go down through the subclasses.

Declaring Our Classes

We can use the Item class we already declared as our base class. We'll add a few members to it (weight and description) and leave it where it is, nested inside our good old MyScript class. We'll also cut out any of the old code we had lying around, giving us a fresh start with no constructor and no methods:

```
public class MyScript:MonoBehaviour
{
    class Item
    {
        public string name = "Unnamed Item";
        public string description = "Undescribed item.";
        public int worth = 1;
        public bool canBeSold = true;
        public int weight = 0;
    }
}
```

This is our base class – the least specific kind of item. We declare our variables that all items have, and we give them default values. Later, we'll be giving them constructors, so the default values shouldn't be necessary because they should always be set by a constructor, but it won't hurt to have them anyway.

Let's declare the class for Equipment – which will serve as the upper type for Weapon and Armor. We'll put it in the same code block that Item is nested in, just under the Item class. Of course, this means it's not a child of Item, but it's a sibling – they're both nested in the same block:

```

public class MyScript:MonoBehaviour
{
    class Item
    {
        public string name = "Unnamed Item";
        public string description = "Undescribed item.";
        public int worth = 1;
        public bool canBeSold = true;
        public int weight = 0;
    }

    class Equipment:Item
    {
        public int currentDurability = 100;
        public int maxDurability = 100;
    }
}

```

Notice the syntax is pretty much the same as any other class declaration, except that after “class Equipment” we have the colon “:” to designate that our class will inherit another; then we provide the name of the class we want to inherit from, which is “Item”. That little part of the declaration is all we need to make Equipment inherit the members of Item.

Next, let’s declare Armor. While we’re at it, let’s exhibit some of what we learned previously and declare an enum called `ArmorType`, which differentiates between the different kinds of armor (gloves, helmet, etc.). Place both of these definitions **within the MyScript code block, just under the Equipment class**:

```

enum ArmorType
{
    Helmet,
    Chest,
    Gloves,
    Girdle,
    Boots
}

```

```
class Armor:Equipment
{
    public ArmorType type = ArmorType.Helmet;
    public int defense = 1;
}
```

The enum declaration is pretty self-explanatory, and we've already gone over that previously.

The Armor class inherits from Equipment, which in turn inherits from Item. This creates a chain. Ultimately, Armor inherits all the variables declared in Item, as well as those declared in Equipment. It adds its own members, one of which stores an instance of the ArmorType enum, which we default to Helmet, and an int for the defense rating of the armor, which we default to 1.

Now we can declare our Weapon class. Again, put the code **within the MyScript code block, beneath the Armor class**:

```
enum WeaponType
{
    Sword,
    Axe,
    Hammer,
    Staff
}

class Weapon:Equipment
{
    public WeaponType type = WeaponType.Sword;
    public int minDamage = 1;
    public int maxDamage = 2;
    public float attackTime = .6f;
    public bool dealsBluntDamage = false;
}
```

This is the same deal as the Armor class. We declare a WeaponType enum with some basic weapon types in it. We then declare the Weapon class, which inherits Equipment and adds some of its own members.

Constructor Chaining

Now we need to add constructors to all these classes so we can neatly create instances that apply the proper values to all these members we've set up. You can probably imagine how painful it can be to declare a unique constructor applying the members for each class, considering each of our lower types will need to have the same code to apply values held in the upper types. We'd have to declare parameters and apply their values for each member declared in `Item`, like `worth` and `name`, in every single lower type as well.

Luckily, we have a tool to make this easier, called **constructor chaining**. The constructors of lower types can “chain” into the upper type constructor, which can in turn chain into its next upper type, and so on. This chaining is pretty much just calling the upper constructor and supplying the parameters on the spot, right in the declaration of the constructor.

Think about it like this: every constructor must declare the parameters of all its upper types, but that doesn't mean we have to apply them all ourselves when we already have these upper constructors that could do it for us. So every constructor will declare all the necessary parameters, including the ones for members of its upper types. But any parameters that are not specific to that lower type will simply be “passed up” the constructor chain, letting the upper constructors handle their assignment.

Let's see it in action. First, let's write our `Item` constructor. **Write this code within the `Item` class code block:**

```
public Item(string name,string description,int worth,bool canBeSold,int
weight)
{
    this.name = name;
    this.description = description;
    this.worth = worth;
    this.canBeSold = canBeSold;
    this.weight = weight;
}
```

This is pretty much your average constructor definition, as we learned about before. Because the parameters are named exactly the same as the members we declared in `Item`, we use “this.” before the member name when applying the value.

Now, let's declare the Equipment constructor. Of course, it goes **in the Equipment class**:

```
public Equipment(string name,string description,int worth,bool canBeSold,
int weight,int maxDurability):base(name,description,worth,canBeSold,weight)
{
    //Apply max durability:
    this.maxDurability = maxDurability;

    //Make current durability match max durability:
    currentDurability = maxDurability;
}
```

Now things are getting bulkier. The first thing you'll probably notice is the `“:base”` coming after our parameter list. This is constructor chaining. The keyword `“base”` refers to the upper class, the one that we are inheriting from – in this case, it's `Item`. We then have a set of parentheses afterward, which is pretty much equivalent to calling the upper constructor – the constructor of the base class `Item`.

When we call the upper constructor, we pass in all the first parameters that this constructor declares, which are all the same as those declared in the upper constructor. These are the parameters that are not specific to `Equipment`. They belong to `Item`, so we give them to the `Item` constructor. We already declared that constructor to apply all those values earlier, after all – and as programmers, we'd hate to repeat ourselves.

We also declare our own extra parameter in this constructor, `“maxDurability”`. This parameter is for a member that's specific to `Equipment`, so we don't pass it into the `Item` constructor. `Item` doesn't concern itself with durability. We use this parameter in the body of our constructor, to apply the value.

You'll notice we only have `maxDurability`. We didn't make a parameter for `currentDurability`. We apply the `maxDurability` parameter value first (`“this.maxDurability = maxDurability”`), and then we simply set our `“currentDurability”` to the given `maxDurability` value so the weapon starts out with maximum durability by default. Remember, `currentDurability` can be accessed simply by typing its name because it's an instanced member of the class that our code is nested inside. And since we don't have a parameter named `“currentDurability”`, we don't need to type `“this.”` before it.

If you want, you can mix some line breaks in wherever you want in the constructor declaration. It'll still do the same thing. For example, you could put a line break before

the “:base” to visually separate the constructor declaration from the base constructor call. It’s a style thing. If it makes reading it more comfortable for you, go ahead and do it.

Now let’s make a chaining constructor for Armor:

```
public Armor(string name,string description,int worth,bool
canBeSold,int weight,int maxDurability,ArmorType type,int
defense):base(name,description,worth,canBeSold,weight,maxDurability)
{
    this.type = type;
    this.defense = defense;
}
```

This is the same concept as before. We have all the same parameters in the same order as the upper types provide them. We can just copy-paste those over. We then chain the constructor to pass all those parameters up the hierarchy. This time, since Equipment is involved, we also add maxDurability to the base constructor call. We have extra parameters for the two members associated with Armor (type and defense), and we apply them in the constructor body.

To give you an overview of the way the parameters of each constructor are passed up the chain, here’s a look at each constructor with the new parameters in bold:

```
public Item(string name,string description,int worth,bool canBeSold,int
weight)

    public Equipment(string name,string description,int worth,bool
canBeSold,int weight,int maxDurability)

        public Armor(string name,string description,int worth,bool
canBeSold,int weight,int maxDurability,ArmorType type,int defense)

            public Weapon(string name,string description,int worth,bool
canBeSold,int weight,int maxDurability,WeaponType type,int
minDamage,int maxDamage,float attackTime)
```

Any parameters which are being “passed up the chain” to be handled by the upper type’s constructor will be in normal text, while those which are handled by that specific constructor are bold. The indention also shows how the classes inherit from each other.

Now the assigning of `currentDurability` is automatically handled for `Armor` and will be for `Weapon` too. If we weren't chaining our constructors, we'd have to copy-paste all this code around, making a messy situation that's dangerously easy to become inconsistent if we ever need to make a change.

Moving on, let's declare a `Weapon` constructor. You could practically do this one yourself at this point, although we will be doing a little special something with the `"dealsBluntDamage"` member:

```
public Weapon(string name,string description,int worth,bool canBeSold,int
weight,int maxDurability,WeaponType type,int minDamage,int maxDamage,float
attackTime):base(name,description,worth,canBeSold,weight,maxDurability)
{
    this.type = type;
    this.minDamage = minDamage;
    this.maxDamage = maxDamage;
    this.attackTime = attackTime;

    //Set dealsBluntDamage based on weapon type:
    if (type == WeaponType.Sword || type == WeaponType.Axe)
        dealsBluntDamage = false;
    else
        dealsBluntDamage = true;
}
```

This time, we declare a parameter for all the `Weapon`-specific members except for `dealsBluntDamage`. And again, we chain the constructor, just as we did with `Armor`, to pass up the parameters that aren't specific to `Weapon`.

Rather than specifying whether the weapon deals blunt or sharp damage by a parameter, we want to just automatically determine that based on the setting provided for the `WeaponType`. We use a simple "if" block which effectively reads "if the weapon type is `Sword` or the weapon type is `Axe`." If so, we set `dealsBluntDamage` to false – the weapon deals sharp damage, or whatever you want to call it. If not, we know the weapon will be of type `Hammer` or `Staff`, so we set `dealsBluntDamage` to true.

Okay, that's the last of it. Now we have all our constructors and data set up for `Item`, `Equipment`, `Armor`, and `Weapon`.

Subtypes and Casting

When dealing with classes that inherit from each other, we often need to refer to them by an upper type when storing them and then figure out what lower type they are on the spot and react accordingly.

For example, it's fine for us to store our player's equipped armor as `Armor` references and their weapon as a `Weapon` reference, but if they have an inventory full of items, it can store any sort of item, so it would look at them merely as instances of `Item`.

When you do this, you can store any subtype of `Item` in those references and simply get it as an ambiguous pointer to an `Item`. You can access their `Item`-specific members like `name`, `description`, `worth`, and so on, but if you want to access members of a lower type like `Weapon` or `Armor`, you must first **typecast** the reference to the expected type.

A typecast is how we tell the compiler what we expected a type to be. It can then look at a reference to some generic type, like `Item`, as a more specific type, like `Weapon`.

Take this following code, where we create a `Weapon`. We provide some generic and unimportant values for its parameters, but the important part is that we store it in a local variable of type `Item`, not `Weapon`:

```
void Start()
{
    Item item = new Weapon("Rusty Axe", "A beat-up rusty axe.", 4, true, 8,
        40, WeaponType.Axe, 4, 9, .6f);
}
```

If you're wondering why we've written an "f" at the end of that last parameter value (".6f"), we'll get to that in a little bit.

The reason we can store a `Weapon` in an `Item` variable is because `Weapon` is a lower type of `Item`. A `Weapon` is more specific but can still be summed up as an `Item` because it has all the same members, even if it has some extra ones tacked on as well. This wouldn't be allowed the other way around – we can't store an instance of `Item` or `Equipment` in a variable of type `Weapon` or `Armor`, for example.

This is because when we reference a `Weapon` or `Armor` instance, we expect them to have all the members associated with those types. If they're instead storing a less specific type, that's just inviting unsavory errors. This is why our compiler won't let us do it in the first place. Part of the reason we use strongly typed languages is to enforce these rules upon us, to keep our code good and clean and to stop us from doing things we probably shouldn't even be doing in the first place.

Now that we've stored our Weapon instance as an Item reference, let's try getting it back to a Weapon reference and see what happens. We'll add a line of code declaring a local variable of type Weapon, and we'll assign the Item value to this variable:

```
void Start()
{
    Item item = new Weapon("Rusty Axe", "A beat-up rusty axe.", 4, true, 8,
        40, WeaponType.Axe, 4, 9, .6f);

    Weapon weapon = item;
}
```

Save and check Unity, and you'll see an error saying this:

Cannot implicitly convert type 'MyScript.Item' to 'MyScript.Weapon'. An explicit conversion exists (are you missing a cast?)

What we're trying to do here is mentioned in the error message: implicitly converting a type.

Converting types can be done **implicitly** or **explicitly**.

The difference is simply in whether we, as the programmer, have manually ordered the conversion. In this case, we haven't used any special syntax to tell the compiler "I want to convert this type to that other type." So that makes this an implicit conversion, because if it's going to happen, it's going to happen without us necessarily telling it to.

These errors sort of act as guards set up to prevent us from accidentally, unknowingly doing type conversions that maybe shouldn't be done in the first place. The compiler doesn't know if that Item stores a Weapon. We may know because we just declared it, but compilers aren't in the habit of making assumptions, even when the context is just one line of code above the error.

We must make an explicit conversion by casting the type. This is an on-the-spot conversion that happens at runtime (meaning while the game is playing).

There are two ways to make this conversion. They each do the same thing but behave a little differently in the case where the types aren't actually compatible as we're expecting them to be.

The first way is to write the name of the type you want to cast to, wrapped in a set of parentheses, right before the "item" reference:

```
Weapon weapon = (Weapon)item;
```

This method **will throw an exception** at runtime if the types aren't compatible. Otherwise, it converts the given object to the type in the parentheses.

The second method is with the **“as” operator**:

```
Weapon weapon = item as Weapon;
```

This method **will not** throw an exception if the types aren't compatible. Instead, it simply returns null, which is the equivalent of a reference that points at nothing. If it succeeds, it returns the type we expect, which is `Weapon`.

Of course, a failure in this case will usually result in an error afterward anyway, because you're likely going to go ahead and use the “weapon” variable as if it's actually storing a weapon. You'll reach in and try to grab some data from it or run a method in it, and since it's null, an error will be generated – just a different kind of error.

Number Value Types

I promised I'd explain what the “f” means when it's placed at the end of a number value, for example, “.6f”. It's known as a **suffix**. A suffix can be tacked onto the end of number values to denote what value type we want the number to be stored as. So far, we've learned of “int” and “float”, but there are a variety of different types which have differing limitations on how high or low a number they can store. Other number types exist which either store a larger range of numbers but take up more memory or store a lower range and take up less memory. For example, an “sbyte” is a much smaller “int”. Where “int” can store a value over 2 billion at the highest (and negative 2 billion at the lowest), an “sbyte” can store no lower than -128 and no higher than 127. As a result, an sbyte takes up less space on the computer, but it's not going to be able to store a high enough value in many situations.

On top of that, there are “unsigned” versions of data types which can't store a negative value (their lowest value is 0), but as a result can store twice as high a positive value. For example, the “s” in “sbyte” means “signed.” An unsigned version of the same type is just “byte.” It stores a value between 0 and 255. There's also an unsigned version of int: “uint”. It can go over 4 billion, but still can't go under 0.

Some of these types have a suffix you can use to easily write a number out as that type, like “f” to make a float. Some don't, and you have to use an explicit conversion to make them a certain type, for example, “(byte)120” to make a byte instead of an int.

By default, a value with no fraction will be an “int”, unless it stores a value outside the range of an int, in which case it goes up to the next largest type that can store the value. A value with a fraction will store a “double,” which is twice as large as a float, but generally more accurate with its fraction value. If a parameter expects a float value, but we pass in a number like .6, we’re really giving it a double. That will give us an error, so we tack that “f” suffix on the end to make it a float instead, fixing the error.

For most purposes, “int” and “float” will serve you perfectly well, and pretty much all of the built-in methods in the Unity engine will expect one of those two. Should you ever need to use another value, however, Table 11-1 gives a rundown of the values for integer types (no decimal value) and their suffix, if any.

Table 11-1. *Integer value data types and their associated suffixes*

Data Type	Value Range	Suffix
sbyte	–128 to 127	--
byte	0 to 255	--
short	–32,768 to 32,767	--
ushort	0 to 65,535	--
int	–2,147,483,648 to 2,147,483,647	--
uint	0 to 4,294,967,295	U
long	–9,223,372,036,854,775,808 to –9,223,372,036,854,775,807	L
ulong	0 to 18,446,744,073,709,551,615	UL

On top of that, there are three different value types we can use for number values with a decimal – also known as “floating point” values, which is what “float” is short for:

- **“float”** uses the **“f”** or **“F”** suffix.
- **“double”** is the default when no suffix is used, although you can also use **“D”** or **“d”**.
- **“decimal”** uses the **“m”** or **“M”** suffix.

Doubles are called double because they’re double the size of a float. A decimal doubles the size yet again, making them four times the size of a float.

To make a somewhat complicated topic short and sweet, floating point values are not totally accurate all the time, particularly when storing high values in them. You might set their value to something and then get the value back, and it's slightly different, off by a little fraction. Data types which are larger than "float" can store larger values and remain more accurate throughout. Again, "float" will likely serve you just fine in most situations you'll encounter.

Type Checking

In the test cases we've dealt with so far, it's obvious to us what types we're dealing with, so we don't really have to worry about errors. But often, you'll first need to verify whether the type of a reference is actually what you think it is before you interact with it.

Say you've got a reference to an Item. For the sake of explanation without getting into a whole other set of problems, let's just assume we have an Item reference spat at us by some code that manages our inventory, and we need to check what lower type it is to determine what sort of functionality it should have.

There are several ways of doing this. Assuming we have a variable or parameter "item" of type Item, how do we check if it's a Weapon or Armor?

One method is with the **"is" operator**. This takes some value on its left-hand side and a direct reference to a type on its right-hand side. If the value is either exactly the same as the type or is a lower (more specific) type of it, the operator returns true. Otherwise, it returns false:

```
if (item is Weapon)
    Debug.Log("Item is a weapon.");
else if (item is Armor)
    Debug.Log("Item is an armor piece.");
else if (item is Equipment)
    Debug.Log("Item is some kind of equipment, but not Armor or Weapon.");
```

In this example, we log one message depending on whether the "item" is a Weapon or Armor, and if it's neither, but is a piece of equipment, we have a generic message to log instead.

Another method to check if the Item reference is a more specific type would be to use the "as" operator we demonstrated earlier to assign the item to a new variable of that

more specific type and then simply do an “if” to test if the result is null. If it was null, then we know the item is not that type. If it was not null, the item was that type:

```
Weapon weapon = item as Weapon;

if (weapon == null)
{
    //Cast failed, 'item' is not a Weapon instance
}
else
{
    //Cast succeeded, we can proceed to use 'weapon' reference
}
```

Sometimes, you might want to check if the instance is exactly the type you’re comparing it with, not a more specific lower type. In this case, the code looks a little whacky compared to our other examples. You call the instance method “GetType” on your object to get a reference to its exact type. You can compare that type with the “==” operator to another to see if they exactly match. But when referring directly to a type like this, you must wrap the type name between the parentheses of “typeof(...)”, as shown in the following:

```
if (item.GetType() == typeof(Equipment))
    Debug.Log("Item type is exactly Equipment.");
```

When we declared “item”, we assigned a Weapon instance to it. Since we’re checking to see if the item is exactly Equipment, this will return false and the message won’t be logged, because while the Weapon is technically Equipment in that it’s a more specific, lower type, it’s not exactly Equipment.

Using these three methods, you can cover pretty much any situation you might come across where you need to check a type.

Virtual Methods

One final aspect of inheritance is the concept of **virtual methods**. We’ll only get into the theory here, not the syntax. We won’t use virtual methods until further into the book, so we’ll wait until then to write them ourselves. For now, let’s learn what they are and what their purpose is, while we’ve still got inheritance on the brain.

You can mark methods as **virtual**, which means that they can be overridden by lower types so that the lower type can tack on its own functionality or even completely overwrite the upper type's functionality.

An example of the purpose of this might be if we had a few extra classes for item types. Let's say we had classes `Consumable:Item` and `Food:Consumable`.

The `Consumable` class is meant to represent things like potions or other such items that can be "used" to consume them on the spot for some effect. It has a virtual method declared inside it called `Use`. This method takes a "target" parameter pointing at a specific entity in the game – a player or an NPC. When the player or an NPC uses a potion, they call `Use` and provide a reference to themselves as the target. The virtual method will determine what happens to the target.

We can then make some classes like `HealthPotion:Consumable` and `ManaPotion:Consumable`. We override the virtual method `Use`, declaring a new version of it in each of these classes. Using the supplied "target" parameter, each implementation of the virtual method can do its own thing. The health potion will restore the target entity's health, and the mana potion will restore their mana. They each provide their own definition of `Use`.

We could then make a `Food:Consumable` which overrides `Use` to simply decrease the target's hunger value by some member variable "float tastiness" specific to the `Food` class.

Then, given a reference to a `Consumable`, we can simply call `Use` on any target, without concerning ourselves with what exact lower type the consumable is. The correct method override will be used automatically – the food will sate hunger, and the potions will restore health or mana based on their type.

Summary

You've now been primed on the fundamental concept of inheritance. It's a somewhat vast topic, and there are some pieces we haven't covered yet, but you hardly need to know all the intricacies of inheritance to be able to code some simple games. As we continue, we'll exercise these concepts bit by bit and grow more comfortable with them.

CHAPTER 12

Debugging

When we set up our code editor back at the very first chapter, we downloaded the “Debugger for Unity” extension in Visual Studio Code. Debugging is a fancy feature of code editors that allows us to mark any line of code as a **breakpoint**, causing the game to pause when we reach that line of code. Once the game is paused, we can hop over to our code editor and see the values of our variables, frozen in time, and even play out one line of code at a time while keeping the game paused.

This can be immensely useful when our code isn’t doing what we intend it to do and we want to figure out why. It’s especially useful when dealing with subtle errors that can’t really be visually studied, such as in situations where you’re just modifying data that you have no way of viewing. A common alternative approach, back before we had proper debugging built into our code editor, would be to start calling `Debug.Log` to give yourself a view of data that might not be what you expect it to be. This can be sufficient in some situations, but is often quite tedious.

With debugging, we can just drop a breakpoint at the problem area, and when that code is reached, the game pauses and we can view the values of all of our variables. For example, if we break during an Update call for a script, we can view instance variables for that script itself (ones declared in the script class) as well as local variables declared within the Update method. We can then continue through the execution of the program one line of code at a time, even going inside the calls of other methods to watch their execution one line at a time.

Setting Up the Debugger

There are a few simple steps to take to make our debugger work with our running instance of the Unity editor. First, if you never installed the Debugger for Unity extension in Visual Studio Code, head back to Chapter 1 to do that. After that, we only have a few clicks to make before we’re ready to debug. Take note that things may look a little

different in your Visual Studio Code compared to the figures shown in this chapter, since you may be using a different color theme.

In Visual Studio Code, click the Debug and Run button on the left side of the window, shown in Figure 12-1, or use the hotkey Ctrl+Shift+D.



Figure 12-1. *The Debug and Run button*

This will open up the left sidebar with controls for debugging. At first, it will likely look like a bunch of nothing, shown in Figure 12-2.

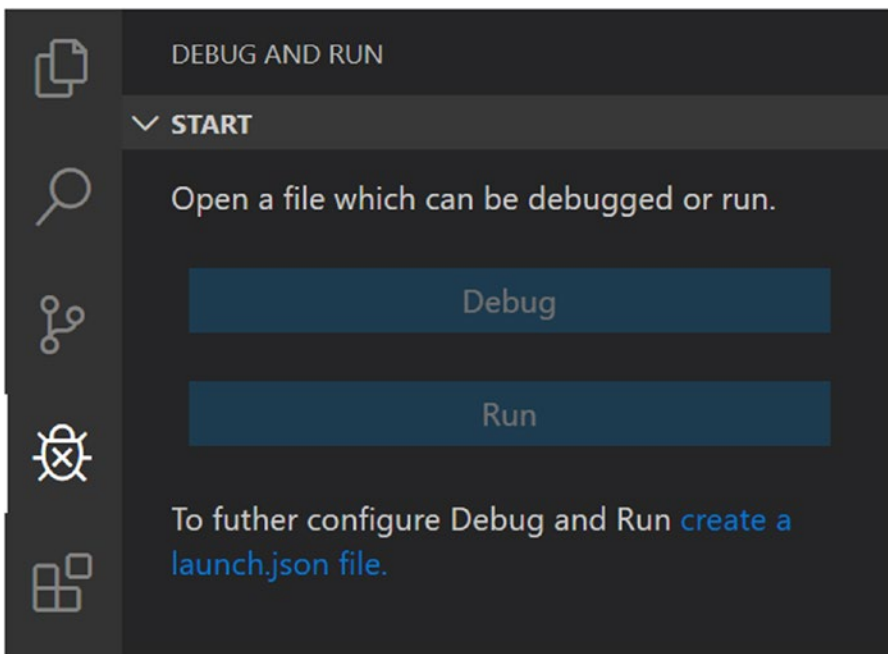


Figure 12-2. *Initial appearance of the Debug and Run sidebar*

This is because we don't have the "launch.json" file that is mentioned. Click the blue link "create a launch.json file," and you'll be prompted to select an environment, shown in Figure 12-3. Select Unity Debugger.

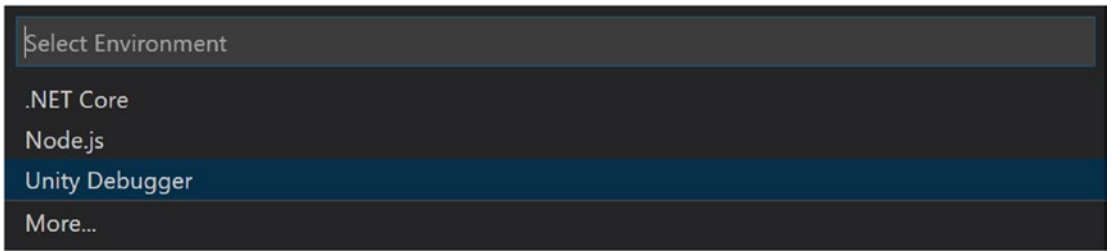


Figure 12-3. *Environment prompt box*

This will create a file, predictably named “launch.json,” and open it automatically. It’s a simple little text file that will be stored in your project folder, but not in the Assets folder, so we won’t see it in the Project window. It just stores some information that the program will use to set up and control the debugger. We don’t need to make any changes to the file, so you can safely close it and forget it exists.

The Debug and Run sidebar will change now that we have made this file. At the top of this window, we’ll have a dropdown field that should read “Unity Editor” and a green play button, shown in Figure 12-4.

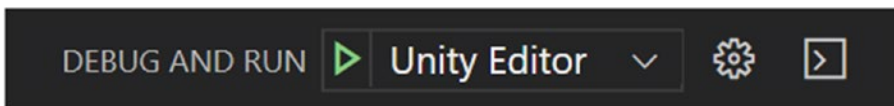


Figure 12-4. *Top of the Debug and Run sidebar after we’ve created our “launch.json” file*

Hitting that play button will start the debugger and attach it to the Unity editor. You can also use the hotkey F5 to start the debugger.

Breakpoints

Now that you know how to start debugging, let’s learn how to place a breakpoint to stop our program when a certain line of code is about to run. The simplest way is to click the empty space at the left side of the line of code you want to break on, just to the left of the line number. You can also press F9 to create a breakpoint at the line of code that your text cursor is currently placed.

Once you’ve set a breakpoint, a red dot will appear at the left side of the line of code. You can click the dot again or press F9 again to remove the breakpoint.

Let's do a quick example to demonstrate debugging features. Create a new script named `DebuggingTest` and add an instance of it to a `GameObject` in your scene. We won't be paying attention to the `GameObject` or anything in the scene, so it doesn't really matter what you put it on.

We'll give the script a simple `Start` event that declares a local variable and then changes its value three times:

```
public class DebuggingTest : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        int a = 5;
        a += 5;
        a *= 2;
        a = 0;
    }
}
```

Add a breakpoint to the `"int a = 5;"` line, which should make it look like [Figure 12-5](#).



Figure 12-5. A breakpoint is added to the line of code, symbolized by the red circle at its left side

Now, with an instance of the script attached to any `GameObject` in your scene, start the debugger with the `F5` hotkey or the green button we mentioned before. Then go to the Unity editor and start playing.

The `Start` call will immediately occur, and that breakpoint will cause the game to be suspended in the Unity editor. Heading back to code, the line that we placed our breakpoint on will be highlighted, signifying that the execution of the code has reached that point and is now paused. The line of code will run next, but not until we tell it to.

In the `Debug` and `Run` sidebar, the `Variables` section will show us all variables accessible from this point in the code, including the `"a"` variable we just declared. It also shows `"this"`, which is the keyword we can use to reference the class that's running the method. Since `"this"` is a script and has many members inside it to view, it has a little

arrow at its side, signifying that it can be “unfolded” to view its contents. Click that, and we can see the members that come with our script, shown in Figure 12-6.

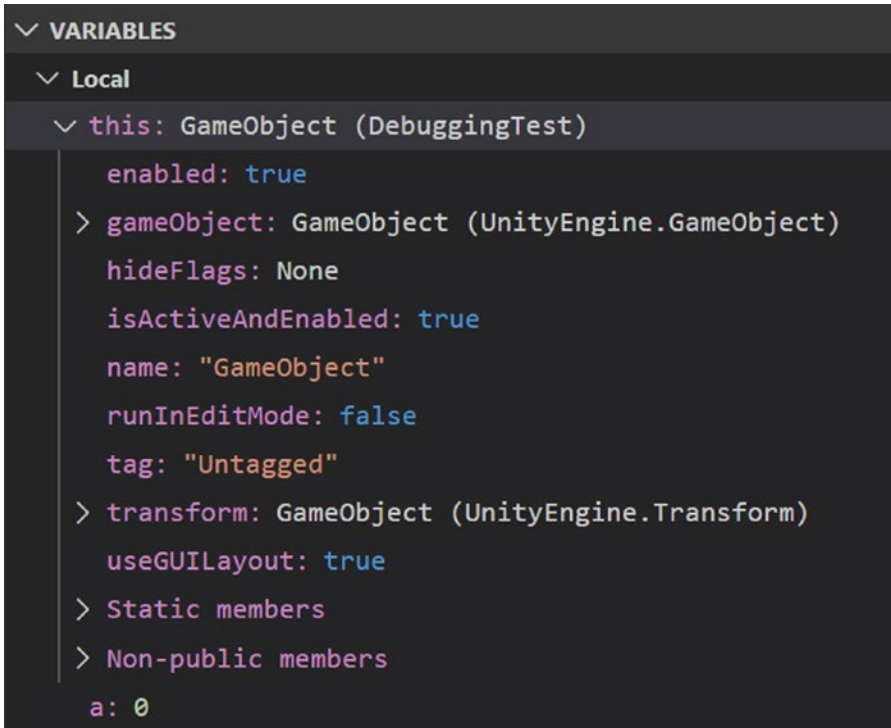


Figure 12-6. The Variables box within the Debug and Run sidebar after our breakpoint is hit. The “this” variable is unfolded to show its members

Some of those members include the `GameObject` the script is attached to and the `Transform` component, each of which has its own members that can be viewed if we were to unfold them as well.

Notice that our “a” variable claims to have a value of 0. This is because the breakpoint will suspend the program before its line of code runs, not after, so our “`int a = 5;`” has not assigned the value of 5 to the variable yet.

Now that the program is suspended by a breakpoint, we have several options available on what to do next. These options are accessed with the little panel of buttons at the top-right corner which appear as soon as you begin debugging, shown in Figure 12-7.



Figure 12-7. *Debugging control panel*

The cluster of six little gray squares at the leftmost side can be clicked and dragged to move the panel around, if you want it somewhere else instead. Disregarding that, from left to right, the buttons are as follows:

- **Continue** (hotkey F5) – The program is resumed until another breakpoint occurs, and we can go back to Unity and continue playing the game.
- **Step Over** (hotkey F10) – Runs the current line of code and then pauses execution again.
- **Step Into** (hotkey F11) – If there are no method calls in this line of code, this operates the same as Step Over. However, if the line of code does call a method, the program will “step into” that method instead and suspend again at the first line of code within that method.
- **Step Out** (hotkey Shift+F11) – Executes whatever is left of the current method and then suspends again. If you accidentally step into a method when you didn’t mean to, you can use this to hop back out.
- **Restart** (hotkey Ctrl+Shift+F5) – Restarts the debugger. This won’t restart the game, though. You have to do that from the Unity editor. You likely won’t need this button when debugging with Unity.
- **Stop** (Shift+F5) – Stops and detaches the debugger so that breakpoints no longer suspend the game.

These controls can be very useful in certain circumstances. Sometimes you want to see how a method ends up returning what it does, in which case you can step into it and watch it execute one line at a time. After, you’ll be brought back to the original line of code that called the method. If multiple methods are called throughout a line of code, you can step into each one individually, one after the other, as they are executed. You might do this, for example, if your breakpoint was on a line of code that called a method, but also called many other methods to return values for the parameters of that method:

```
SomeMethod(A() + B(), C(), D());
```

In that case, you could step first into A, then B, then C, then D, and then finally the `SomeMethod` call.

But enough chatter about the theory of it all. Press **Step Over** once. The highlighted line of code will move forward to the next line. In the **Variables** section, you should see the value of “a” update to become 5. Press **Step Over** again to run the “a += 5” line, and, as expected, the value updates to become 10. Run “a *= 2” and the 10 becomes 20. You get the point.

Once we run out of code to run, **Step Over** will stop having any effect, but the program will remain suspended, so you’ll have to use **Continue** to unfreeze the game when you’re ready.

As a final note, notice that none of the controls actually skip code. The code will still be executed – the controls simply dictate how much of it to execute before stopping again.

Using Unity’s Documentation

Debugging is a means of finding out what the cause of a problem is on your own. A large part of being a programmer is problem solving: using the tools, you have to figure out what’s going wrong or how to go about doing something. While we’re on that topic, let’s learn a little bit about how to find helpful resources as a programmer.

I can tell you how to do this and that throughout the course of this book, and you can learn a lot, but eventually, you’ll have to strike out on your own and figure out how to do something yourself. And that’s what can make or break a programmer. A large part of your learning is probably going to come when you start programming your own stuff, when you’re learning what to do as you go, putting the pieces together to achieve something.

When faced with a challenge to overcome, you need information on what you’re dealing with. Any decent search engine will often return Unity’s official documentation as one of the first results. Just search for “unity” followed by the class name or component type you want to learn more about, like `Light` or `Rigidbody`. Unity separates its documentation into two forms – the **Manual** and the **Scripting API**:

- The **Manual** documentation is more of an instructional page describing how a component works and how one might use it, usually with images and examples. It is aimed at teaching a new concept to the reader.

- The **Scripting API** is technical documentation specifically for programmers. It describes classes and their members, such as variables, properties, and methods. You can navigate to a page devoted to any of these individual fields, which will give further information – notably, a description of the field’s purpose, but also important information like what type a variable stores, what type a method returns, the types and names of parameters that a method expects, all of the overloads of a method and how they differ from each other, and so on. It is aimed at programmers looking for information about Unity’s built-in types and methods and usually provides code samples of how to use and interact with individual members or types.

The latter form of documentation is commonly found in various programming environments. If you ever stray away from Unity and into a different game engine, there will likely be API documentation with a similar structure and layout, showing you types and their members. For example, Microsoft – the company behind the C# programming language – maintains similar documentation for the many built-in types offered with the language. Want to learn how to save and read from files on the user’s computer? That sort of stuff is in the `System.IO` namespace. Search for it, and you’ll probably get the official Microsoft documentation as one of the first results. There you can find the classes and methods you’ll use to get the job done, including descriptions and code samples. If you don’t have someone to tell you where to find the relevant classes, just search for a more general term, like “how to read from a text file in C#”. Follow the trail until you know what to write!

Good API documentation like this can make a huge difference on your experience with coding in an environment. You might hit a point where you don’t really want to read a tutorial on how to use a component – you just want descriptions of the types you’ll be working with and the fields they store. With proper documentation, you can get this with a simple search and be on your way.

Summary

In this chapter, we learned how to run our debugger and work with its controls. While it may seem redundant right now, a debugger can be a lifesaver, allowing you to look at the process your code is taking one line at a time as it executes. Being able to read the values of variables at each step in the code can do much more to illuminate a problem than `Debug.Log` calls. We also learned how Unity's documentation is organized and how we might go about finding information ourselves should we need it.

PART I

Obstacle Course

CHAPTER 13

Obstacle Course Design and Outline

Now that you've been introduced to the basics of the Unity engine and the fundamentals of programming, we're going to develop our first example project. You might not feel totally confident with all the things we've learned so far. Maybe some things have slipped through the cracks. That's okay. More and more of it will stick around as you put the concepts to use.

Before you get too excited, note that the example projects we develop won't have flashy graphics and audio. We're focusing on the code that runs them, not on making or importing art and audio assets. This way, you can focus on your own domain until you've got it down. Once you've finished this book, you can branch out depending on what your goals are. A lot of independent game developers will learn to do a little of everything, but you could also find assets on the Internet – some are even free – and use those to give yourself more time to spend mastering programming. Or you could just pair up with some like-minded individuals who want to make a game with you and have them work on the art and audio for you.

Developing a game can be a daunting task. A lot of moving parts and pieces go into the creation of a game. Things may change while you're developing a project and putting those parts together – something may have seemed like a good idea only for you to find out it kind of sucks once you've seen it in action. It happens, but it's no excuse to go in blind.

There's a fine line between planning too little and too much, and you might never get it totally right. You might plan for all sorts of stuff that never comes about because it turns out to not be feasible – you can't implement it or you run out of time or any other number of excuses. Some developers just don't plan much at all and let the pieces fall in place. You'll find your middle ground with time.

But before we dive in, we're going to do a little bit of planning. It'll make it easier to get things how we want it the first time, so we don't find ourselves doubling back and redoing things.

Gameplay Overview

Let's go over how the game will play. It will use an overhead camera, pointing down at the player character. The player will move with the WASD keys and press Space while holding a movement key to perform a quick dash in that direction. The dash has a short cooldown before it can be used again, but otherwise has no cost to performing it.

The game levels will be blocked out with cubes for walls, colored differently than the floor to distinguish them. The player will collide with these walls on touch, confining them to the playing space we've set up.

The objective of the game is to avoid touching obstacles within that playing space. Each level has a starting point where the player begins and a goal to reach at the end, which is just a simple, circular podium. Dying at the hands of the obstacles will respawn the player at the beginning of the level until they reach the end (or give up and quit out of frustration).

Obstacles will include

- **Patrolling hazards** which move from one point to the next in a series of points, return to the original point when finished, and then repeat. We will set the points up ourselves, individually, for each patrolling hazard we place in a Scene. As hazards, they kill the player on touch.
- **Projectiles** which are fired by obstacles, traveling in a straight line until they hit a wall. They are also hazards that will kill the player on touch.
- **Wandering hazards** which are confined to a single rectangular area, occasionally choosing a new random point to go to. Upon choosing a new point, they gradually rotate to face that point and then begin moving toward it (so as not to unpleasantly surprise the player in unpredictable ways).
- **Spike traps** which periodically raise a circle of hazardous spikes and then lower them down again. The player must cross when the spikes are down.

The main menu will allow the player to select which level they want to play on. They can cycle through the levels to load them and view them from overhead. When ready, they press a button to begin playing the level.

From within a level, the player can press Escape to bring up an in-game menu, which allows them to return to the main menu if they do not wish to play the level anymore.

At first, we'll implement the important stuff. Game developers often focus first on the core mechanics, which will determine whether the game is fun or a dud. We hardly need any proper menus and level selection if the game isn't fun at the core, right?

We'll keep it confined to one scene while we implement these core mechanics, like the player movement, the obstacles, and the player dying and respawning.

We'll worry about proper level selection and whatnot after we get our major features functioning.

Technical Overview

To prepare for our project, we should outline what we expect to have to implement and consider how these things will be implemented in our engine. With experience, this becomes easier, but even if you're a beginner, it doesn't hurt to at least try to figure out the general idea behind implementing each feature before you start on any of them. While going over everything, you might spot things you hadn't thought about yet or flaws in the way you thought you might do something. Having a high-level overview of the whole project going through your head at once can be useful in that way.

Player Controls

The player will be represented by a cube with another little cube on the top, colored differently and pointing along their local forward direction (otherwise, you won't be able to tell which way they're facing). This is shown in Figure 13-1.

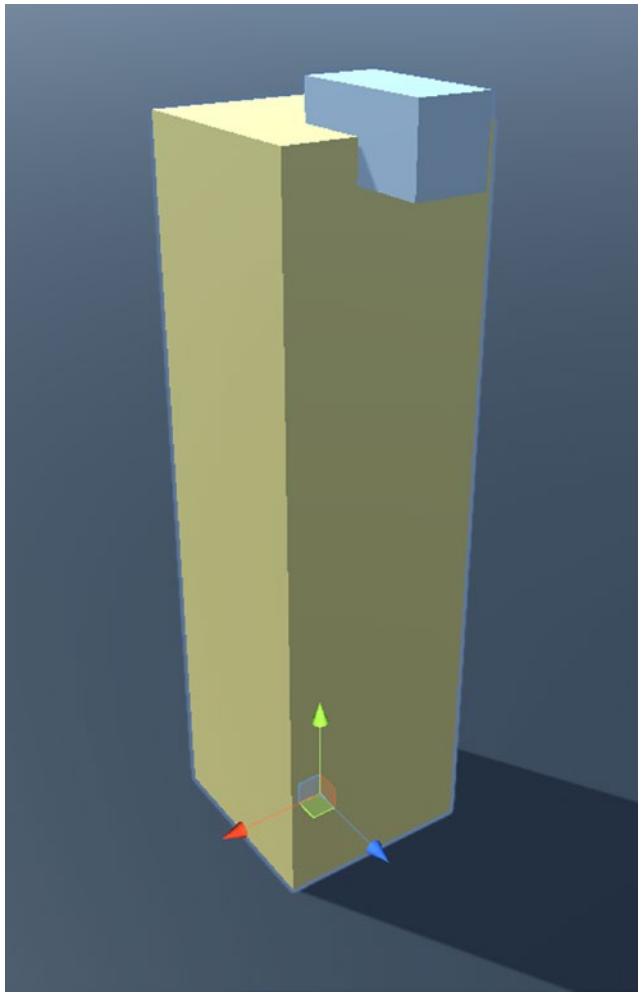


Figure 13-1. *The player, facing toward the camera. The blue arrow gizmo at the bottom shows the local forward axis of the player*

The Player script will be on the root GameObject, which we'll base off an empty GameObject. This will hold separate GameObjects for the cubes that visually represent the player. When we rotate the player to face toward their last movement direction, we only rotate the cubes, not the root player GameObject.

This way, we can make the camera a child of the player so it moves with them, and since the root itself isn't rotating, the camera won't rotate with it (which would be very awkward, unintended, and jarring).

We'll have a Player script that handles all the player-related functionality. Primarily, this means movement and dodging.

The movement will be done either with WASD keys or the arrow keys, since some users prefer one over the other. We'll put a little bit of momentum to it. The player will take a bit to reach full speed and to lose full speed. We won't go too crazy with this, though, as we risk making the player too slippery. We want to keep things pretty accurate to avoid frustrating accidents because our character was skidding around when we wanted them to stop, but we still want a little bit of springiness so the movement doesn't look or feel too jerky.

Dodging will be done by pressing space, providing a fast lunge in the direction the player is facing. It'll be over quick, but during that time, the normal movement won't be calculated so that only the dodge velocity goes through.

Death and Respawn

When the player dies, we'll wait a short duration and then return (or "respawn") them at the start of the level. While they're dead, we don't want to let them move around, so we'll disable the Player script until the respawn has finished.

Levels

All we really need to shape a level is just a plane underneath the player acting as the floor with its own unique color, and then we'll place raised blocks and size them however we need to block out the "walls" of the level. When the player moves, they'll collide with the walls, stopping them from leaving the play area. The game will be viewed from above with an orthographic camera, which is a camera that doesn't use perspective – sort of like viewing the world in a 2D way. This way, you don't really see the walls as raised, but rather, they just look like a different color among the rest. We won't see the sides of walls (since there's no perspective), and we won't have shadows or lighting.

This is a somewhat hacky way of designing levels, but first and foremost, it'll be functional and quick to produce. If we were making a game we planned on publishing, we'd be dealing with art assets and designing our levels in some other way, but we aren't focused on such things.

To allow the player to win the level, we'll have a Goal script that detects when the player walks (or slides, I should say) over it, returning them to the level selection screen when they do.

Level Selection

Each level will have its own scene dedicated to it. The main scene will be loaded by default when the user starts the game. Our menu will allow the player to cycle through levels, which will load in the scene for the desired level so we can preview it. When we load in a new scene, the last one will be cleaned up and removed for us automatically.

Each level scene will have a preview camera in it, which we'll position over the level to give the player a preview of its entirety.

Once we begin playing a level, we'll switch to the player camera and disable the preview camera.

Obstacles

We went over a similar concept for our obstacles in an early chapter, when we were first learning about components. We'll make these scripts:

- **Hazard**, which makes a `GameObject` “kill” the player on touch
- **Wanderer**, which gives a `GameObject` the erratic “wandering” movement we described
- **Patroller**, which lets us set up a path for a `GameObject` to repeatedly move along
- **Shooting**, which makes a `GameObject` periodically fire projectiles
- **SpikeTrap**, which makes a hazardous `GameObject` rapidly spring outward and then lower harmlessly back into its spot, waiting to raise again after a specified wait duration

By mixing these scripts on `GameObjects`, we can implement all the obstacles we desire.

Project Setup

Let's get a new Unity project ready for this game. Open Unity Hub and create a new project with the New button in the top-right corner. We've been here before, so you might remember the window that pops up. Select the 3D template, name your project `ObstacleCourse`, and save it to whichever folder you want.

In your Project window, navigate to the Assets folder, where you'll see there's already a folder named Scenes inside it. Add some new folders to give it a little company:

Materials, Prefabs, and Scripts.

While we're at it, let's rename the default SampleScene asset (located in your Assets/Scenes folder) to "main". This can be done by right-clicking the asset in the Project and then selecting "Rename". After confirming the rename, Unity might pop up a window asking you if you'd like to reload the scene. Go ahead and do that.

The scene will have a Directional Light and a Main Camera in it by default. We can leave those be, but we also want to add a basic floor for the player to stand on. Using the top-left menus, select **GameObject** ► **3D Object** ► **Plane**. Select the Plane if it isn't already selected, and navigate to the Inspector to set it up:

1. Name it "Floor".
2. Set the Transform position to (0, 0, 0) to ensure that it is centered at the world origin.
3. To make it large enough that we shouldn't have to worry about running out of space, set its scale to (1000, 1, 1000).

Your Project view should look like Figure 13-2 when you're done.

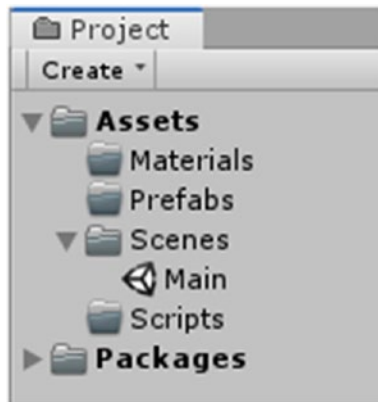


Figure 13-2. Our Project window once we've added our folders and renamed the scene

Summary

We've established how our project is expected to play: a simple top-down obstacle course where the player moves around with the WASD keys and avoids dangerous obstacles while attempting to navigate to the end of the level. Next, we'll start implementing the pieces one at a time.

CHAPTER 14

Player Movement

With our new project in place, let's start coding our player's movement. We're going to get a little fancy with it to exercise our programming skills and learn about some new methods.

As we mentioned in the last chapter, we want movement to be fluid – not jerky, but not slippery, either. A little bit of time to build to the maximum speed will make it less of a sudden jerk when the player begins moving. Likewise, we want some time to lose speed after you stop holding down the movement keys. It won't be much, because a game like this requires tight control of the character, so we definitely don't want the player feeling like they're skidding around.

We'll handle the player's movement velocity frame by frame. The velocity is a vector (an object with X, Y, and Z values) that depicts the change in position the player will be constantly experiencing from their movement. We use units per second as our measurement. For example, if velocity is (15, 0, 12), then the player will be moving right at 15 units per second and forward at 12 units per second. They won't be moving on the Y axis, because the player has no up or down movement – just forward, back, right, and left. When playing the game, you might think to describe the Z axis as “up and down,” because that's the direction it will take your character on the screen, but it's actually forward and back. Remember, the camera is positioned overhead, pointing down at the player, so if the player were to actually go directly up in world space, they'd be going toward the camera.

To move the player with the sort of smoothness we're after, we'll be increasing or decreasing the velocity based on the player's input and constantly moving them by that velocity. The WASD and arrow keys will both work for movement, so the player can use whichever they prefer. While holding the keys, the velocity will change gradually to produce the desired movement.

While the player is moving, we'll point their model in the direction their velocity is taking them. To visualize this, the player is given a simple model made of a few cubes, one of which pokes out along the player's local forward axis.

Player Setup

Let's set up our player GameObject. We'll be staying in the same "main" scene for now, using it as a playground of sorts to develop and test whatever we're working on at the moment.

The hierarchy will look as pictured in Figure 14-1 – we'll walk through the process of creating each GameObject in a bit.



Figure 14-1. A view of our Player in the Hierarchy

The base GameObject named "Player" is the Transform that we'll actually move. Of course, this means the children will move with it. But we don't rotate this Transform, because it holds our camera. Remember that children will act as if they are physically attached to their parent, so if we spin that base Player GameObject around, the camera will swivel around with it. This will be very jarring and disorienting for the player.

That's why we create an empty GameObject named Model and place it inside the Player. This is the "model holder." It's just there to store all of our model-related GameObjects. Our script will reference this Transform and rotate it to face the movement direction. This way, the base GameObject (which holds the camera) is never rotated. Only the model is.

The model holder simply has two cubes inside it, one named Base and one named Top.

Here's what you'll do to create it:

- Create an empty GameObject (Ctrl+Shift+N on Windows or Cmd+Shift+N on Mac) and name it Player.
- Create an empty child of this GameObject (while selecting the Player, Alt+Shift+N on Windows or Opt+Shift+N on Mac) and name it Model. Ensure that its local position is (0, 0, 0) so that it's positioned exactly on the Player.

- Create a cube that's a child of Model by right-clicking Model in the Hierarchy and selecting 3D Object ► Cube. Name this cube Base. Set its local position to (0, 2.5, 0) and set its scale to (1.4, 5, 1.4). This is local scale, but all of its parents have a scale of 1, so it's equivalent to world scale. The local Y position is half the Y scale, to keep the bottom of the cube at the pivot point instead of the center (the pivot point being the position of the Player GameObject).
- Create a second cube, this time a child of Base. Name it Top; set its position to (0, .5, .5) and its scale to (.33, .1, .7).
- The Main Camera GameObject should already be present in the scene. Drag it onto the Player GameObject, making it a direct child of the player root (not the Model!). For now, we'll give it a local position of (0, 24, -5) and rotation of (70, 0, 0) to put it a good way over the player's head, back a little bit, and tilted down. Figure 14-2 shows how the camera Transform should look in the Inspector.
- Create a prefab for the Player by dragging the Player from the Hierarchy window to the Prefabs folder in the Project window.

The Top cube will poke out along the forward axis of the player (the facing direction). This way, we always know where the player is pointing.

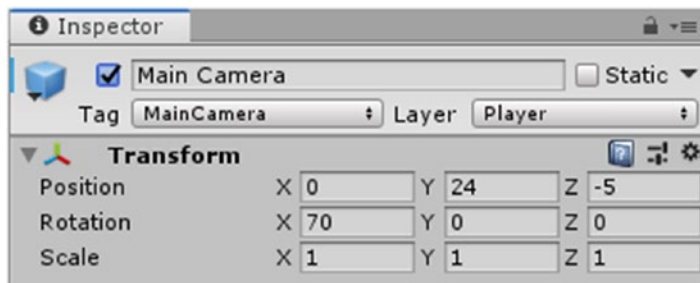


Figure 14-2. The Transform component of our Main Camera in the Inspector

Materials and Colors

Now we want to color the cubes, each one a different color so the Top stands out from the Base. To do this, we use **materials**. A material is a built-in Unity asset that stores information about how a mesh (or a 2D image) should be rendered. Most notably, they allow you to apply **textures** to meshes. **Textures** are pretty much normal 2D images that are stretched over and wrapped around a mesh (3D object). This is how we turn objects from flat surfaces that render only a single color to things like rock, tree bark, tile floors, plaster walls, and so on. Some games use pictures of real things; others use stylized drawings.

Materials have many different settings, but they relate mostly to integrating art assets into your game project. As we've said before, we're not really going to dig into that over the course of our book. We'll use materials simply to apply color to our solid objects, which is one of the most basic ways to use a material.

In the Project view, create a material named Player Base and another one named Player Top, both within your Materials folder. With either one of the new materials selected, the Inspector will show a somewhat daunting list of fields we can edit for the material. Luckily, we only need one little control. Near the top, just beneath the bold text "Main Maps," there's a swatch of color to the right side of the word "Albedo" (see Figure 14-3), with an eyedropper icon beside it.



Figure 14-3. The color field for a material is the rectangle with an eyedropper icon beside it

This is a color field for the material. By default, it's a shade of white. Click that rectangle to pop up a color window that lets us change the color, shown in Figure 14-4. You can also click the eyedropper and then click a color anywhere on your screen to select that color.

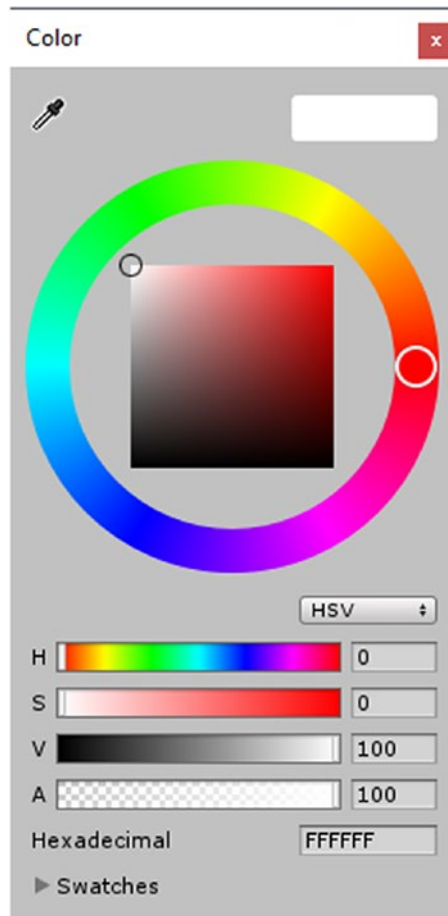


Figure 14-4. *The color popup window showing the default color for a new material*

Let's learn how computers look at color. It's a concept that will likely come up often for a game developer, even if you're just programming. Don't worry, it's not too complicated.

The top half of the color popup window is an interactive wheel with a square in the middle. Click the wheel to change the shade of the color. Click within the square to change how vibrant and bright the color is.

The bottom half of the window lets you set number values directly. The first three fields will resemble individual **components** of the color, showing a single letter on the left side, a colorful bar to the right of it, and then a little box with a number in it. The number is the value of that color component. The letter is short for a word that depicts what the component controls. The bar is a visual representation of how raising or lowering that component value will change the color. You can click within the bar to

change the color component individually. You can also just edit the number directly, if you're savvy like that.

So what are the color components? It depends on the **color model** you're using. There are two models that computers use to depict a color: **RGB** (Red, Green, Blue) and **HSV** (Hue, Saturation, Value). Each model resembles a color in a different way, but they can both be easily converted to and from the other. You can change the color model with the little dropdown field beneath the color wheel.

The **HSV** model tends to be a little easier to understand and predict. **Hue** is the color – red, green, blue, purple, cyan, yellow, and so on. It is a value from 0 to 360, which corresponds to an angle on the color wheel. **Saturation** is a value from 0 to 100, depicting how vibrant the color is. A lower value will make it turn closer and closer to a gray shade. **Value** is the brightness of the color. A lower value will shift the color closer to black. It also ranges from 0 to 100.

The **RGB** model stores the amount of red, green, and blue in the color, either ranging from 0 to 255 or, alternatively, from 0 to 1 (using fractional values). All shades of color can be obtained by mixing the correct proportion of these three primary colors. For example, black is no color (0, 0, 0), white is the maximum amount of each color (255, 255, 255), and yellow is a mix of red and green (255, 255, 0). Colors which use a more even mix of all three components will appear less saturated; colors which use high values in one or two components will be more vibrant, the sort of shade you might see in toys or fancy sports cars. Colors which use a low amount in all components will be darker.

That covers the first three fields, but what about the fourth field, with the letter “A” beside it?

This is **alpha**. It's there regardless of the color model. It determines transparency, and ranges from 0 to 100. Changing it won't do anything because our material itself doesn't support transparency, but just so you know, a low alpha makes the color more transparent (see-through), and a high alpha makes it more solid (also called oblique).

Finally, the bottom field is the **hexadecimal value** of the color. Sometimes shortened to hex value, this is a way to describe a color in a single string of letters and numbers. Change this field, and you'll change all of the color components at once. It is, however, less user-friendly and pretty much looks like an arbitrary slur of characters.

I've chosen a pale yellow for the base and a pale blue for the top. You can exhibit some creativity and choose your own colors if you like, but if you want to have it chosen for you, the hex value for the yellow I'm using is FFEF86, and for the blue, it's B6DAF5. Just type that into the Hexadecimal field in the color window for either material, and you'll get the same color.

Now that you have your materials, you just need to apply them to the correct cube GameObjects. Technically, this is done through the Mesh Renderer component in the Inspector, but it's easiest to just click and drag the material asset from the Project window to the desired GameObject in the Hierarchy or Scene window, which will apply the material to the Mesh Renderer automatically.

And at that, we have our player hierarchy set up, and we even have a colored model. Let's press on and start doing some coding.

Declaring Our Variables

To begin coding our movement, we'll need a script that we have attached to the Player. Create one in the Scripts folder within the Project window and name it "Player". Place an instance of the script as a component on the root GameObject, the one named Player, by dragging the script from the Project window to the Player in the Hierarchy window.

Now, open up your project in your code editor if you haven't already. An easy way to do this is by clicking the Assets button in the top toolbar and then selecting "Open C# Project." If this doesn't bring up your code editor, you probably need to set the editor by navigating through the top bar to Edit ► Preferences ► External Tools and setting the External Script Editor field.

Once you've given Visual Studio Code a moment to open up, you'll need to open the Player script we just made. You can do this by pressing Ctrl+P on Windows or Cmd+P on Mac and then typing "Player" in the popup box that results. This is a quick way to open a file by name, assuming you know what you're looking for.

You can also use Ctrl+Shift+E on Windows or Cmd+Shift+E on Mac to open the Explorer in the left sidebar; then find and click the Player script there to open it.

The first thing we'll do is declare some variables that we'll be using to handle the movement of the player. They'll go right at the top of the class declaration for our script:

```
public class Player : MonoBehaviour
{
    //References
    [Header("References")]
    public Transform trans;
    public Transform modelTrans;
    public CharacterController characterController;
```

```

//Movement
[Header("Movement")]
[Tooltip("Units moved per second at maximum speed.")]
public float movespeed = 24;

[Tooltip("Time, in seconds, to reach maximum speed.")]
public float timeToMaxSpeed = .26f;

private float VelocityGainPerSecond { get { return movespeed /
timeToMaxSpeed; } }

[Tooltip("Time, in seconds, to go from maximum speed to stationary.")]
public float timeToLoseMaxSpeed = .2f;

private float VelocityLossPerSecond { get { return movespeed /
timeToLoseMaxSpeed; } }

[Tooltip("Multiplier for momentum when attempting to move in a
direction opposite the current traveling direction (e.g. trying to move
right when already moving left).")]
public float reverseMomentumMultiplier = 2.2f;

private Vector3 movementVelocity = Vector3.zero;
}

```

You’ve pretty much seen everything we’re doing here already at some point in the previous chapters, except for the declarations “VelocityGainPerSecond” and “VelocityLossPerSecond”. These are *properties*, a concept we haven’t discussed yet. They probably look a little confusing to you right now, but we’ll clear that up in a bit.

The first thing we do is declare references to other components that we plan on using. When you declare a variable of a component type (like Transform), you can set the value of the variable through the Inspector, so we don’t have to write code to find the component we want. We just set the references once in the Inspector, and then we can use them anywhere in our code. We use the Header attribute on the first reference, “trans”, to put a bold title above the references in the Inspector, keeping them in their own little section at the top of the script.

We have a “trans” variable to point to the player Transform, which we do for performance reasons – it’s slightly faster for a script to hold its own reference to its own Transform instead of using the built-in member “transform” each time. We also have a

reference to our model holder transform, “modelTrans”, and a reference to a component we haven’t used yet, called a CharacterController. This is how we’ll be moving our character – we’ll add that component and learn about it later in this chapter.

Moving down, we have a comment “//Movement” and a new header to separate a different set of variables, these ones pertaining to movement. We also use the Tooltip attribute with each variable, which gives us a way to provide a description for our variables. These are neat because they’ll show up in the Unity editor when you hold your mouse over a variable field in the Inspector. Not only do they act something like comments for your code but they’ll also help document the purpose of your variables within the engine. If you’re programming for a team of game developers, this can be useful to help guide non-coders on how to use the scripts you write.

The tooltips offer some insight on what each variable is for, but let’s go over them in one place:

- **movespeed** is the maximum speed at which the player is capable of moving per second.
- **timeToMaxSpeed** is the time taken, in seconds, to build up to the maximum speed.
- **timeToLoseMaxSpeed** is the time taken, in seconds, to go from the maximum speed to stationary when you stop holding down the movement keys entirely.
- **reverseMomentumMultiplier** is used to make it easier for the player to stop traveling in one direction and begin traveling in the opposite direction at a moment’s notice. While working against existing velocity (e.g., if the player begins holding the right arrow, but they were already traveling left), the velocity that we gain is multiplied by this value. So if we want to gain speed twice as fast when working against existing velocity, we would set this to 2. If we want to gain speed 50% faster, we’d set it to 1.5. Conversely, we could make it harder to reverse momentum by making the value lower than 1.0, although we don’t plan on doing that.

These are the movement-related variables we expose in the Inspector. These ones can be changed to alter the player’s movement dynamics if we ever see fit. Once we’ve got some obstacles and game mechanics in place to give us a frame of reference, we may want to change the values a bit.

Lastly, `movementVelocity` is a private variable of type `Vector3`, which is a data type exposed by the Unity engine to resemble a vector of three axes: the X, Y, and Z values. As we've learned already, vectors can resemble various things, from a rotation angle (each axis is between 0 and 360) to a position to a scale. We use it to represent the current ongoing velocity of the player, measured in units per second. The values will range from `movespeed` to negative `movespeed` on the X and Z axes.

Properties

`VelocityGainPerSecond` and `VelocityLossPerSecond` are two examples of our first usage of **properties**. We've declared them as simple means of “folding up” a common math operation to a more understandable name, so we can just point to that name instead of typing out the math operation every time we want to use it. Remember, Don't Repeat Yourself!

Properties can be looked at as some thing of a crossbreed of variables and methods. They are declared much like variables, but they prompt a block of code instead of ending with a semicolon, and they don't get a default value assigned to them. When referring to them and setting them, they are treated like variables: you can get their value, and you can set it.

But when declaring them, you define what gets returned when their value is referenced and/or what happens when their value is set. This is where they behave something like a method. Within the code block of a property, you can declare what we call a **getter** and a **setter**. Simply enough, they are declared with the **get** or **set** keyword, respectively, followed by a code block.

The **getter** is a block of code that is run whenever the property is referenced. It must return a value, as methods do.

The **setter** is a block of code that is run whenever the property is being set to a new value. It does not return a value. Within the setter block, you can access the value being assigned to the variable by typing “value”.

A property can declare a getter, a setter, or both. If only one is declared, you can only perform that operation on the property. That is to say, you can't get the value of a property that doesn't declare a getter, just as you can't set the value if it doesn't declare a setter. In our case, we aren't declaring a setter. We don't plan on setting the value for either of these properties, since we're just using them to give us a shortcut to a common math operation. So we just declare a “get” block and put a single line of code in it that returns the result of the math operation.

It might look a little confusing because of the way it's formatted. As we've depicted before, C# doesn't really pay attention to line breaks and indentation. It isn't part of the syntax. When the code is read by the computer, it's not really using line breaks and indentation to know when statements and code blocks end. Rather, it's paying attention to curly braces and semicolons. It doesn't care if we throw a bunch of line breaks in the middle of a statement, so long as we end that statement with a semicolon like we're supposed to. We just use line breaks and indentation in specific ways to create a consistent and readable standard for code – not because the syntax is forcing us to.

These rules can be stretched and broken at times. We've done that with our property declarations. Since they're just a single, simple line of code in a "get" block, we don't space them out as you would normally space out a property. We declare it all in one line for brevity's sake.

Normally, you might format it like this:

```
private float VelocityGainPerSecond
{
    get
    {
        return movespeed / timeToMaxSpeed;
    }
}
```

You can do it this way if you prefer. Some people might even insist that you space it all out like this, because it's "the proper way." But I think it's less clunky on one line, so I do it like that. If I had a setter as well, or if my getter had more than one line of code in it, I would absolutely spread it out like that last example, but this is a good example of when it's not a big deal to forsake the "rules" of code formatting if you're more comfortable with the result.

Tracking the Velocity

Now we know what sort of variables are available to us in our script, so let's start working on the per-frame update logic that makes the movement actually happen. Most of this logic is about handling the movementVelocity variable – that's the bread and butter, because it depicts how much we move every second.

Before we start typing away in the Update method, one practice you might want to develop is the act of separating different chunks of Update logic into smaller methods.

For example, to neatly separate the movement logic from any other logic we may be running in the Update method somewhere down the road, we can just declare a method “private void Movement()”, write all our movement logic there, and then call that method in Update.

This can be an organizational lifesaver if you ever have a particularly complicated script. Your code editor will help you out by letting you fold up individual blocks of code, so you can easily hide away the code you don’t need to see.

This is what it’ll look like, before we throw down any of the actual movement logic. All of this is nested in the script class block, of course:

```
private void Movement()
{
    //...movement code goes here.
}

private void Update()
{
    Movement();
}
```

We declare Movement as a simple, private method with nothing to return. We then call it in Update. If we implement something new in the Player script later, we can do the same thing, keeping the new set of logic in its own space as well.

Now let’s start filling the Movement method with code. Keep in mind, all of this code is running every frame, so it’ll be running in small but very frequent increments, constantly. And of course, any value that we expect to be applied “per second” must be multiplied by Time.deltaTime as we did before.

First, we’ll use a nested series of “if” and “else” blocks to alter the movementVelocity based on the player input as well as the current state of the movementVelocity (to allow us to apply the reverseMomentumMultiplier when working against existing velocity). After that, we’ll check if there is any movementVelocity on this frame and, if so, use it to move the player.

Let’s get to it. We’ll start with the Z axis to implement forward and backward movement and go over it piece by piece. First, we’ll detect the W key or the up arrow key to handle forward velocity. To cap the velocity so that it never goes over “movespeed”

in either direction, we use simple math methods “Mathf.Min” and “Mathf.Max”. Each method lets us pass two float parameters into it and will return the lowest (Mathf.Min) or highest (Mathf.Max) of the two. Pretty basic, right?

This is a common way to cap a value while increasing or lowering it. It’s a cleaner, shorter alternative to adding to the value and then checking if it’s higher than the maximum to set it back down to the maximum if it is. We just have one line of code that uses Mathf.Min or Mathf.Max to set the value. Instead of increasing or decreasing the value, we set it to the result of the Min or Max call. Here is an example of increasing a value this way:

```
value = Mathf.Min(maximumValue,value + addedAmount);
```

In this example, we set “value” to the result of the Min method. The Min method is given two parameters and will return the lowest of the two. The first one is the maximum amount we want “value” to be capped at. The second is the current state of “value” plus whatever amount we want to add to it. If the result of adding to “value” is lower than that maximum amount, then the result is returned. But if the result goes over the maximum value, we’ll always get the maximumValue instead. Beautiful, isn’t it?

A similar concept can be used when subtracting:

```
value = Mathf.Max(minimumValue,value - addedAmount);
```

Here, we use Mathf.Max instead, and we subtract instead of adding. In this case, if the result of subtracting from “value” is higher than the minimum value, we will get the result. If the result goes under the minimum value, we’ll get the minimumValue instead.

To sum it up, we **use Min when adding to cap at a maximum amount**, and we **use Max when subtracting to cap at a minimum amount**.

Putting the pieces together, we handle forward movement like this:

```
//If W or the up arrow key is held:
if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow))
{
    if (movementVelocity.z >= 0) //If we're already moving forward
        //Increase Z velocity by VelocityGainPerSecond, but don't go higher
        than 'movespeed':
        movementVelocity.z = Mathf.Min(movespeed,movementVelocity.z +
            VelocityGainPerSecond * Time.deltaTime);
```



```

else //Else if we're moving back
    //Increase Z velocity by VelocityGainPerSecond, using the
    reverseMomentumMultiplier, but don't raise higher than 0:
    movementVelocity.z = Mathf.Min(0,movementVelocity.z +
    VelocityGainPerSecond * reverseMomentumMultiplier * Time.
    deltaTime);
}

```

This may look a bit scary at first, but let's break down what's happening. Keep in mind that `velocity.z` will be negative if the player is moving backward and positive if the player is moving forward. If `velocity.z` is “movespeed”, then the player is moving at full speed forward. If `velocity.z` is “-movespeed” (negative movespeed), then the player is moving at full speed backward.

The process goes like this, in plain English:

- If either the W key or the up arrow key is held
 - ...and the player currently has forward (positive) momentum or no momentum at all, then add to the forward momentum, but never let it go higher than “movespeed”.
 - ...and the player currently has backward (negative) momentum, increase and use the `reverseMomentumMultiplier`. Since the velocity is negative, we don't allow it to go over 0. This way, as soon as we hit 0 velocity, we stop using the `reverseMomentumMultiplier`.

Understand this, and you'll be fine for the rest of this chapter. Whenever we change the velocity, we change it using this concept of Min and Max to cleanly cap the value while we assign it.

The amount we're adding to the velocity to increase it is making use of one of the properties we declared earlier: `VelocityGainPerSecond`. Let's take a look at the amount we're adding in seclusion (when not reversing momentum):

```
movementVelocity.z + VelocityGainPerSecond * Time.deltaTime
```

This is fairly straightforward. Thanks to the property, it practically describes itself in plain English: start with Z velocity and add the velocity gain per second. Since it's “per second,” it has to be multiplied by `Time.deltaTime`, as we learned before. Another thing we previously learned: The multiplication operator “*” will take precedence over a

“+” or “-” operator. This means it’s always calculated first, with whatever is on its left and right sides, and then the “+” or “-” will be calculated with the result of that. So we know that `VelocityGainPerSecond` alone is being multiplied by `Time.deltaTime`, not the result of `movementVelocity.z + VelocityGainPerSecond`.

If you want this to be visually clear, you can use parentheses to separate them yourself. Some would argue that it’s not necessary, since it does the same thing, but if it makes it easier for you to read and understand, then go for it:

```
movementVelocity.z + (VelocityGainPerSecond * Time.deltaTime)
```

Moving on, when we’re reversing momentum, we have this:

```
movementVelocity.z + VelocityGainPerSecond * reverseMomentumMultiplier *
Time.deltaTime
```

It’s the same thing, but we throw the “*” operator in there to apply the `reverseMomentumMultiplier`. You probably know that the order in which we multiply things doesn’t count for anything. We could switch the position of any of those latter three references however we want, and the result is the same.

And with that, you’ve learned pretty much all of the concepts you need to learn to understand how this velocity-handling system operates. There are no surprises left – just little variances on what we’ve already done.

Moving on, we can now see how backward movement looks. This code will go immediately after the “if” code block that we just wrote (the one that checks for W or the up arrow key being pressed):

```
//If S or the down arrow key is held:
else if (Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow))
{
    if (movementVelocity.z > 0) //If we're already moving forward
        movementVelocity.z = Mathf.Max(0, movementVelocity.z -
            VelocityGainPerSecond * reverseMomentumMultiplier * Time.
            deltaTime);

    else //If we're moving back or not moving at all
        movementVelocity.z = Mathf.Max(-movespeed, movementVelocity.z -
            VelocityGainPerSecond * Time.deltaTime);
}
```

This is pretty much a copy-and-paste of the forward movement, but with certain little changes made. Notice that it starts by checking for the S or down arrow key, but this time with an “else if”, not a normal “if”. This is to ensure that if both the up and down movement keys are held, only one of them (the up key) will have priority instead of both happening at the same time.

As well as this, we now use Max instead of Min, because we’re subtracting the velocity now.

Let’s put it to plain English again:

- If either the S key or the down arrow key is held
 - ...and the player currently has forward (positive) momentum, then decrease the forward momentum while applying `reverseMomentumMultiplier`, but never let it go lower than 0.
 - ...and the player currently has backward (negative) momentum or no momentum at all, decrease the momentum, but never let it go lower than “-movespeed”.

This leaves one final condition we need to account for: the loss of velocity when neither the forward nor backward key is held. If we don’t implement this, the player will just keep going after releasing the movement keys. That’s definitely not what we want.

We’ll put an “else” after the “else if” that checks if the back key is held, and then, if momentum is positive, we decrease it and cap at 0. If it’s negative, we increase it and, again, cap at 0. If it’s exactly 0, we don’t do anything:

```
else //If neither forward nor back are being held
{
    //We must bring the Z velocity back to 0 over time.
    if (movementVelocity.z > 0) //If we're moving up,
        //Decrease Z velocity by VelocitylossPerSecond, but don't go any
        lower than 0:
        movementVelocity.z = Mathf.Max(0,movementVelocity.z -
        VelocityLossPerSecond * Time.deltaTime);
```

```

else //If we're moving down,
    //Increase Z velocity (back towards 0) by VelocityLossPerSecond,
    but don't go any higher than 0:
    movementVelocity.z = Mathf.Min(0,movementVelocity.z +
    VelocityLossPerSecond * Time.deltaTime);
}

```

This leaves us with nothing to do but copy over these principles and apply them to right/left instead of forward/back. To write this code, we can recycle most of what we've already written. We'll copy and paste the code that handles forward/back movement (everything in the Movement method so far) and then change the hotkeys to A/D and the left/right arrows. We'll also change any reference to the Z axis to instead reference the X axis and edit the comments to keep them accurate.

Be diligent with this! If you accidentally leave a “z” instead of a “x” somewhere, you'll be in for some unexpected behavior:

```

if (Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.RightArrow))
{
    if (movementVelocity.x >= 0) //If we're already moving right
        //Increase X velocity by VelocityGainPerSecond, but don't go higher
        than 'movespeed':
        movementVelocity.x = Mathf.Min(movespeed,movementVelocity.x +
        VelocityGainPerSecond * Time.deltaTime);

    else //If we're moving left
        //Increase x velocity by VelocityGainPerSecond, using the
        reverseMomentumMultiplier, but don't raise higher than 0:
        movementVelocity.x = Mathf.Min(0,movementVelocity.x +
        VelocityGainPerSecond * reverseMomentumMultiplier * Time.deltaTime);
}

else if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow))
{
    if (movementVelocity.x > 0) //If we're already moving right
        movementVelocity.x = Mathf.Max(0,movementVelocity.x -
        VelocityGainPerSecond * reverseMomentumMultiplier * Time.
        deltaTime);
}

```

```

    else //If we're moving left or not moving at all
        movementVelocity.x = Mathf.Max(-movespeed, movementVelocity.x -
            VelocityGainPerSecond * Time.deltaTime);
}

else //If neither right nor left are being held
{
    //We must bring the X velocity back to 0 over time.

    if (movementVelocity.x > 0) //If we're moving right,
        //Decrease X velocity by VelocityLossPerSecond, but don't go any
        lower than 0:
        movementVelocity.x = Mathf.Max(0, movementVelocity.x -
            VelocityLossPerSecond * Time.deltaTime);

    else //If we're moving left,
        //Increase X velocity (back towards 0) by VelocityLossPerSecond,
        but don't go any higher than 0:
        movementVelocity.x = Mathf.Min(0, movementVelocity.x +
            VelocityLossPerSecond * Time.deltaTime);
}

```

Applying the Movement

This leaves us with nothing left but applying the movement velocity so the player actually moves. To do this, we'll use the **CharacterController**. This is a built-in Unity component designed to provide movement and collision detection for a character. If we just moved the player by setting their Transform position, it wouldn't detect collisions (the player would pass through everything). But moving them through a CharacterController component will cause them to bump into and slide against solid objects in their path.

First, let's set up the CharacterController component on the Player GameObject (the same one with the Player script component attached to it).

Add a CharacterController by selecting the Player in the Hierarchy, then navigating to the bottom of the Inspector window and clicking the Add Component button. This will bring up a little dropdown box. Now you can navigate to the CharacterController by clicking Physics and then CharacterController. Alternatively, you can search for a

component type by name: type “Character Controller” into the search bar and press Enter or click the CharacterController listing when it shows.

The CharacterController uses a capsule shape to test for collisions. This is represented in the Scene window as a wireframe set of green lines. That’s how big the player is perceived to be with the current settings of the Controller.

Locate the Center, Radius, and Height members of the CharacterController in the Inspector. For our purposes, we want a Center of (0, 2.5, 0), a Radius of 1, and a Height of 5. This makes the collider mostly cover the Player, with a bit of overhang on the sides so we retain a little bit of personal space. The rest of the settings can be left as is. When you’re done, the CharacterController should look like Figure 14-5 in the Inspector.

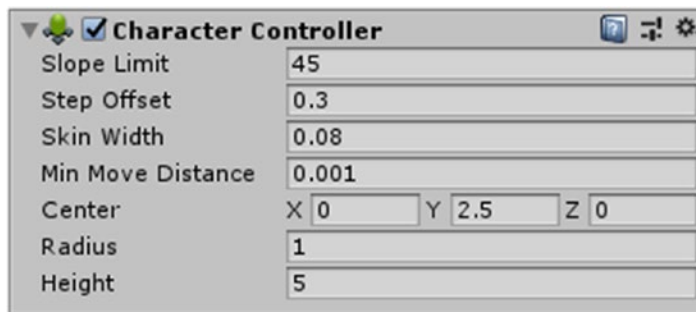


Figure 14-5. The CharacterController component of the Player is shown in the Inspector

With that set up, we can now set the reference to the CharacterController variable that we declared in our Player script. Applying references to components in the editor like this can be done in various ways:

- In the Inspector, click the little circle icon at the right side of the “characterController” field in the Player script. This opens a popup window that lets you select any GameObjects in the Scene that have a CharacterController attached. Click one to set the value.
- In the Inspector, left-click and drag the bold text “Character Controller” that makes up the header of the component. You can then drop it onto the variable field in the Player script to set the reference.

- In the Hierarchy, left-click and drag the Player GameObject itself into the Inspector and drop it on the variable field in the Player script. Unity will recognize that the field wants a CharacterController reference and find that component on the GameObject you're dragging.

Any way will work, and some are more convenient than others at times.

Once the field is set, it will show the name of the GameObject who owns the CharacterController, and after it, it will specify "(Character Controller)" to note the actual Component type that's stored.

While we're at it, we'll also need to set the reference to "trans" and "modelTrans", which should be listed just above the CharacterController field that we just set.

This works the same way. You can drag the Transform component of the player directly onto the "trans" field and then drag the "Model" GameObject from the Hierarchy onto the field in the Inspector.

Now we can head back to our code and apply the movement. At the bottom of the Movement method, past all of the code that handles the input and movement velocity, add this bit:

```
//If the player is moving in either direction (left/right or up/down):
if (movementVelocity.x != 0 || movementVelocity.z != 0)
{
    //Applying the movement velocity:
    characterController.Move(movementVelocity * Time.deltaTime);

    //Keeping the model holder rotated towards the last movement direction:
    modelTrans.rotation = Quaternion.Slerp(modelTrans.rotation, Quaternion.
    LookRotation(movementVelocity),.18F);
}
```

This starts out pretty simple. We check if the X and Z axes have movement: if X is not 0 or (the "||" operator) if Z is not 0. If so, we apply the movement by reaching into our "characterController" reference and calling its method "Move". This method takes a single argument: a Vector3 for the movement desired on this frame. We multiply it by Time.deltaTime to ensure it's "per second" instead of "per frame."

Beneath this, we rotate the model holder to point along the movement direction. This involves working with rotations, something we haven't done much yet, so it comes with a little bit of unfamiliar territory.

Unity represents rotations with a type called **Quaternion**. These are mathematically somewhat complicated (they even *sound* intimidating), but luckily, we don't have to mess with that math. Mostly, you just call built-in methods when working with rotations like this. Just know that a single instance of Quaternion represents a rotation, which is pretty much a direction that something can point toward, or an angle at which something can be tilted. In the Inspector, we look at it as a Vector3, where each value is between 0 and 360, because that's a more user-friendly way of looking at it. But internally, Unity represents rotations as Quaternions.

The built-in methods we're using here are "Quaternion.Slerp" and "Quaternion.LookRotation".

Slerp is a term you'll hear somewhat often in game development, particularly when dealing with vectors and rotation. It is short for "spherical linear interpolation." To put it simply, it's a method that takes three arguments:

- The rotation we start with (a Quaternion)
- The rotation we want to end with (also a Quaternion)
- A float for how fast we want to get there

The float is a multiplier, between 0 and 1, and pretty much means "What fraction of the way to the desired rotation will we be going on this frame?" So if it's set to 0, we'll go nowhere at all; if it's set to 1, we'll get there immediately. But if it's set to something like .18, we'll get 18% of the way there.

If you look at how we use this method, you'll see how this generates a smooth change. Each frame, we set the rotation again, calling the Slerp method to do so. Each call to the Slerp method takes the current rotation as the start and the same target direction as the end. So we'll rotate 18% of the way to the desired rotation on the first frame, then another 18% of what remains on the next frame, and so on. Each frame, the difference between the current rotation and the target rotation becomes smaller and smaller. This generates a comfortable sort of "spring effect" where at first, the rotation is snappier and more pronounced, but as the difference becomes smaller, the rotation smoothly comes to a stop. This will make it look somewhat more visually appealing than rotating by a flat amount each frame.

To get the desired rotation (the second parameter in the Slerp call), we call Quaternion.LookRotation. This is a method that takes a Vector3 parameter and returns back a rotation pointing forward along whatever direction that vector is heading.

To sum up the whole line of code, we pass in `movementVelocity` to a `Quaternion`. `LookRotation` call to get a `Quaternion` pointing us forward along the direction our movement is taking us. We use `Slerp` to take our rotation toward that one by `.18x` the difference per frame.

If we didn't want a smooth effect, we could do this instead:

```
modelTrans.rotation = Quaternion.LookRotation(movementVelocity);
```

That will directly set the model rotation to jerk our player to face along the movement direction immediately. It won't look terrible, but it will be jerky – particularly when going from a standstill to flip in the opposite direction.

With that said, we've implemented our movement. It looks nice and it feels nice. At last, we could now play our game and see it in action. Make sure your `Player` script is saved since the last change.

You should be able to move your player around with either the WASD keys or the arrow keys. The camera should stay in a fixed position above the player, moving with them as they go (but remaining in the same local position). The player model will smoothly rotate themselves to point at the current movement direction.

Summary

In this chapter, we gave our player the ability to move. We learned how to create and apply basic materials to add some color to our `GameObjects`, how to use the `Mathf.Max` and `Mathf.Min` methods to cap values as we raise or decrease them, and how to handle our player's velocity per frame so they gradually pick up and lose speed based on their input with the WASD keys. We also exercised some foresight in how we wanted our `GameObjects` to be set up in the Hierarchy to make sure our model rotated separately from the rest of the `Player`. This keeps the camera in the same location even as the player model turns to point along the movement direction. Some other key points to remember are as follows:

- A **property** is much like a variable, but it defines code blocks for exactly what happens when we **get** or **set** the variable.
- When **adding** to a number value, use **Min** to cap the resulting value below a maximum amount.

- When **subtracting** from a number value, use **Max** to cap the resulting value above a minimum amount.
- The **Slerp** method can be used to move one value toward another by a fraction of the difference between those two values.
- A **Quaternion** is a data type that resembles a rotation. The “Transform.rotation” member is a Quaternion that resembles a Transform’s current facing direction.
- The **Quaternion.LookRotation** method takes a Vector3 as its parameter and returns a Quaternion rotation to point a Transform toward the direction that vector travels.

CHAPTER 15

Death and Respawning

In this chapter, we'll be implementing a bare-bones system to allow us to "kill" our player (don't worry, it's very nonviolent) and send them back to the spawn point.

We'll set the spawn point when the game first starts, so that wherever we place the player in the scene, that's where they'll respawn. We won't be doing anything flashy for now. When the player dies, their model will disappear, and the Player script will be disabled to keep it from updating. After a few seconds of wait (to punish them for failing), we'll move them back to the spawn point and enable them again.

To start, we'll declare these variables in the Player script class, underneath the movement-related variables we declared before:

```
//Death and Respawning
[Header("Death and Respawning")]
[Tooltip("How long after the player's death, in seconds, before they are
respawned?")]
public float respawnWaitTime = 2f;

private bool dead = false;

private Vector3 spawnPoint;
```

Since these variables relate to a different system, we'll start them with a new Header attribute to keep our Inspector looking tidy. We declare a float that we can edit in the Inspector if we want to, depicting how long the player must wait before they are revived.

We also declare two private variables for our own uses, which we don't need to be setting in the Inspector:

- **dead**, a bool depicting whether the player is alive (false) or dead (true) at the time
- **spawnPoint**, a Vector3 for the location at which the player will respawn

Enabling and Disabling

To set our spawn point, we'll declare a `Start` method. If you recall, this is a built-in Unity event for scripts (the `MonoBehaviour` class we inherit from). Declare it, and Unity will automatically call it for us when the script is initializing, before any `Update` calls occur.

This would be a good time to mention that there are two similar built-in event methods you might use to do this: **Awake** and **Start**. They both run only once in the lifetime of a script component (unless you call them yourself, in your own code). There are a few subtle differences, though:

- When a scene first loads in, all `Awake` calls will always happen before any `Start` calls happen. If one script should always be initialized before another, you would use `Awake` on that script and `Start` on the other.
- `Awake` will happen as soon as the script is initialized, whether it's enabled or not. On the other hand, `Start` will happen on the first frame that the script is enabled.

If the initialization code you're writing needs to occur as soon as an object is spawned, but that object is going to be inactive when it first spawns, then you'd want to use `Awake`. Aside from that, you'd generally want to use `Start` if you don't need `Awake`. If you use `Start` by default with all your scripts, then you have the option of using `Awake` to easily make a script initialize before the others if you need to sometime down the road.

We haven't used the concept of enabling and disabling scripts or `GameObjects` much yet. We're going to learn about that and make use of it in this chapter.

By default, `GameObjects` are active, meaning that all of their components will run and update. We can deactivate them through our code as well as in the Unity editor itself, by selecting the `GameObject` and unchecking the box just left of the `GameObject` name at the head of the Inspector (see Figure 15-1).

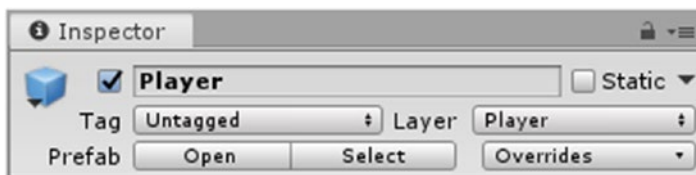


Figure 15-1. The head of the Inspector showing the `Player` `GameObject`. The checkbox to the left of the `Player` name can be unchecked to deactivate the `GameObject`

Disabling a `GameObject` stops all of its components from running. That includes renderers, collision detection and other physics-related components, lights, cameras, and so on. It also means all of the children of that `GameObject` won't be active, either. Deactivated `GameObjects` will have gray text for their name in the Hierarchy to make it obvious that they're not active. Since the children of an inactive `GameObject` are also deactivated, their names will be gray as well (see Figure 15-2).



Figure 15-2. *An inactive `GameObject` beside an active `GameObject` in the Hierarchy*

This goes for components too, and that includes script components. Each component which supports enabling/disabling will have a checkbox beside its header in the Inspector (see Figure 15-3).



Figure 15-3. *Folded `Player` script component and `CharacterController` component in the Inspector. The checkbox to the left of the component name can be unchecked to disable the component*

Components are enabled by default, but we could disable them in the editor if we only want them to begin operating at some later point in the game, when our code enables them.

Some components and scripts won't have this checkbox because they don't have any consistent logic that can be turned on or off on a moment's notice. Your scripts won't have the checkbox if they don't declare any of the built-in events that are affected by enabling/disabling the script, like `Start` and `Update` – but not `Awake`, because as mentioned, it doesn't care if the script is enabled or disabled. It happens either way, as soon as possible.

For the most part, you'll probably get off fine using `Start`, but this is a nice distinction to know about if you ever get into some edge case where you want `Awake` instead.

As mentioned, deactivating a `GameObject` will deactivate all of its children too. But each `GameObject` has its own, individual state depicting whether it is active or disabled. Even if a `GameObject` is active in and of itself, it could still effectively be disabled because any of its parents are inactive.

Unity retains the state of each individual `GameObject`, but its “actual state” is dependent upon the state of its parents as well. This means if you have a child that’s deactivated, and then you deactivate and later enable its parent, the child remains deactivated – it stores its own state.

Unity gives us the ability to make this distinction ourselves in our code. With a reference to a `GameObject`, we have two separate bool variables we can access to check the `GameObject` state:

- **`GameObject.active`** is the `GameObject`’s current, effective state, which will be false if any of its parents are inactive and true if its parents are active and the `GameObject` itself is active.
- **`GameObject.activeSelf`** is the `GameObject`’s individual state. Even if this is true, the `GameObject` could still be inactive because any of its parents are inactive. To put it simply, this state is only used if all parents are active; if any of the parents are not active, this is ignored, and the `GameObject` will be inactive regardless.

Technically, these are properties, like those math properties we made for movement in the last chapter. They only allow us to get them, not to set them. To set the active state of a `GameObject`, we need to use a method.

We’ll do that in a second, but for now, let’s write that `Start` method and set our spawn point. It’s just one line of code in the method:

```
void Start()
{
    spawnPoint = trans.position;
}
```

That’s pretty self-explanatory: as soon as the script is first enabled, we set `spawnPoint` to the `Player`’s `Transform.position`. After this, we won’t be changing the spawn point.

Death Method

Now we have our spawn point. Let's make a method that kills the player when it's called. We'll make it public, and we'll have it return nothing (void):

```
public void Die()
{
    if (!dead)
    {
        dead = true;
        Invoke("Respawn",respawnWaitTime);
        movementVelocity = Vector3.zero;
        enabled = false;
        characterController.enabled = false;
        modelTrans.gameObject.SetActive(false);
    }
}
```

First, we make sure we avoid an awkward case where the Die method somehow gets called when the player is already dead, with a simple “if” that checks if we’re alive before proceeding.

We then immediately set “dead” to true, since we plan on being very dead by the end of this method. We use a new concept, **Invoke**, which we haven’t used yet before.

Invoking methods is a means of calling a method after a given wait period. First, we give a string for the name of the method we want to call: “Respawn”. Then, we give it a float for the number of seconds we want to wait before the method should be called.

We haven’t declared this Respawn method yet, but we will after. It’s important that the names are exactly the same. Even if you just mess up the capitalization, for example, by naming the method “respawn” instead of “Respawn”, it won’t be called. Unity will, however, log a message in the Console for you if it fails to invoke the method, to let you know why it’s not working.

When you invoke a method, you can’t pass any parameters into that method. It only follows that the method must also be declared with no parameters as well. If it has parameters, the invoke will fail.

After the Invoke call, we want to make sure the player loses all of their momentum when they die, so they don’t have any left when they respawn later. The movement velocity is reset to (0, 0, 0) using a bit of shorthand: Vector3.zero. It’s just a slightly shorter

way to type “new Vector3(0,0,0).” It does the same thing. Technically, Vector3.zero is just a property in the Vector3 type. It has a getter, but no setter, and always returns a new Vector3(0,0,0).

Then we set “enabled” to false. This is an instance variable that every component type has, depicting whether or not the component is enabled. Scripts are components too. Set it to false, and your script stops running. It’s as easy as that. This doesn’t interrupt the code that’s currently running, it just prevents built-in events like Update from occurring until the script is re-enabled. Our invoked method will still happen even though the script is disabled.

We do the same for the CharacterController. Since this is the component that’s catching collisions for the player, if we don’t deactivate it, the player will still be there as far as physics are concerned, so they’ll still be touched by anything that might pass by.

For GameObjects, we have to call the SetActive method and pass in the value we want to set it to: true or false. We go through the modelTrans to access its GameObject. Remember, a Transform is just a more specific version of a Component, and all Components (including scripts) have a “gameObject” member pointing to the GameObject that they’re attached to. Using this, we don’t need to declare a separate reference to the model GameObject.

Respawn Method

The Respawn method is already being invoked, so now we just need to declare it. It will set “dead” back to false, place the player back at the spawn point, and enable the script, CharacterController, and model again. This reverts the player back to a normal, functioning state, ready to try the level again:

```
public void Respawn()
{
    dead = false;
    trans.position = spawnPoint;
    enabled = true;
    characterController.enabled = true;
    modelTrans.gameObject.SetActive(true);
}
```


That pretty much does it for our simple death-and-respawn system. Now, when a hazard touches the player, all we need is a reference to the Player script, through which we can call the public method `Die`. We'll handle that in the next chapter, but to test if the method works beforehand, we can throw down some simple, temporary code that lets us kill the player on the spot with a hotkey. Since you're an expert programmer now, you can write this up in no time, right? Just write this code in the `Update` method:

```
if (Input.GetKeyDown(KeyCode.T))
    Die();
```

Save, run the game, move away from the initial point, and press T. Your character should vanish, and since we deactivate the model instead of the root Player GameObject, our camera doesn't get deactivated with it. After the wait time, which is 2 seconds by default, you should pop back up at the spawn point and be able to move around again.

One final measure we could take would be to reset the player's rotation when they respawn. It's a little odd to see your character respawn facing the same angle. This won't be hard at all. You can probably guess how to do this yourself, but I'll tell you how to add it anyway.

Beneath your `spawnPoint` variable, declare this variable:

```
private Quaternion spawnRotation;
```

Set that variable to the `modelTrans.rotation` **in your `Start` method**:

```
spawnRotation = modelTrans.rotation;
```

Finally, apply the rotation to the `modelTrans` somewhere **in your `Respawn` method**:

```
modelTrans.rotation = spawnRotation;
```

Now, if you ever want the player to spawn at a particular rotation, you can rotate their model to point that way, and it'll stick throughout the game. You can vary this for each level, if you ever need to.

Once again, the reason we're dealing only with the Model instead of the root when it comes to rotation is because the camera is a child of the root, so if we ever rotate the root, the camera pivots around it, which changes the direction the camera faces. Since all of our movement is in world directions, none of it is going to change to match the new camera rotation.

You can try this for yourself, if you want to see what I mean in action. Apply some Y rotation to the Player in the Inspector – 90 degrees, let’s say. Then play, and try moving around. The W or up arrow key will now take you to the left side of your screen, and the S or down arrow key will now take you to the right side of your screen. It’s no fun, right?

Just switch that Y rotation back to 0 when you’re done playing, and everything will go back to normal.

Summary

In this chapter, we set up our player’s death and respawning. Now we’ll have a method to call to “kill” the player when we’re coding our hazardous obstacles. Some key points to remember are as follows:

- Deactivating a `GameObject` will stop all of its attached components from doing whatever it is they do. As well, all child `GameObjects` will be considered inactive.
- You can deactivate a `GameObject` in the Scene by selecting it and unchecking the box to the left of its name in the Inspector header.
- **Awake()** and **Start()** are both built-in events you can declare in a script to run the code within as soon as the script is loaded and ready, one time only for each script instance.
- All **Awake** calls will occur **before** any **Start** calls.
- **Awake** calls will occur even if the script or its `GameObject` is inactive. They might be inactive by default in the Scene, or they could be a prefab that’s inactive, which means it will still be inactive when it’s first created. Either way, the **Awake** calls are still going to go off as soon as possible.
- **Start** calls do not occur on an inactive script or `GameObject`. They will wait until the script first becomes active before they are called.
- The **Invoke** method can be used to call a function by name on the script after a given wait duration.

CHAPTER 16

Basic Hazards

Now that our player can be killed, let's start coding up some villainy to kill them. We'll make a generic script called Hazard that makes a `GameObject` kill the player on touch. We'll use this to create dangerous projectiles and an obstacle type that periodically fires them.

Collision Detection

Before we make a script that reacts to a touch, we'll need to dive into collision detection to understand how Unity deals with such things. There are three major concepts that relate: colliders, Rigidbodies, and layers.

Colliders are components that, when added to a `GameObject`, give it an invisible shape that will interact with other colliders. There are various types of colliders that provide different shapes: spheres, boxes, capsules, and even a mesh collider that can add collisions to a complex mesh shape.

You'll notice that the physical objects we've created thus far, like the floor we created under the player (a plane) and the cubes that make up the player model, all have collider components of some sort on them. A plane will have a Mesh Collider, a cube will have a Box Collider.

I didn't mention it before to avoid treading this ground too early, but we don't actually need any colliders on our player model itself. We already have the `CharacterController` providing collisions for the whole Player object. So you can go ahead and remove the Box Collider components from the Base and Top `GameObjects` that make up the player model, just to tidy them up. You can do this by clicking the little gear icon on the top right of the component listing in the Inspector and then clicking "Remove Component," shown in Figure 16-1.

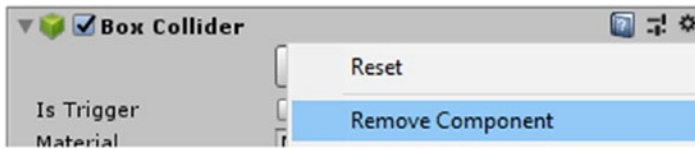


Figure 16-1. After clicking the little gear in the top-right corner of a component header in the Inspector, the “Remove Component” option can be selected

A collider can be marked as a **trigger collider** via a checkbox field in the Inspector. Trigger colliders don’t act as physical objects, allowing other objects to pass through them – however, they still send certain event messages to attached script components, which lets us use them to detect when an object enters an area or gets near a switch or something to that effect.

A **Rigidbody** is a component that adds physical behavior to a `GameObject`. When a `Rigidbody` is attached to a `GameObject`, it is realistically affected by collisions – for example, it will be pushed and rotated by other objects hitting it or pressing against it. It also adds gravity to the object and enables you to add force (motion) and torque (twisting and rotating) to the object through scripting.

To enable collisions, a `Rigidbody` still needs a collider – that’s what defines its shape in the physics engine. An object with a `Rigidbody` but no collider can have gravity and can have forces applied to it, but it will pass through other objects.

If you’re using a `Rigidbody` like this, you would not directly move or rotate the `Transform` through your code. Rather, you’d add force and torque to it through a reference to the `Rigidbody` component. Interfering with the `Rigidbody` by changing the `Transform` directly could cause unwanted results, since the Unity engine is simulating the physics itself and doesn’t account for any sudden movements you cause to the `Transform`.

Nothing in our game is going to require such realistic physics control. What we’ll be doing to move things like projectiles, where we directly move the `Transform`, is known as **kinematic movement**.

`Rigidbody`s can be marked as **kinematic** through the “Is Kinematic” field in the Inspector. This will override these behaviors that move and twist the object. If you want to have finer control over an object, like a player character, you would mark it as kinematic. If it’s kinematic, you’ll apply movement to it through the `Transform`, but it will still trigger collisions in other objects.

The general rules of thumb are

- If it moves, use a Rigidbody. If it remains still, don't use a Rigidbody.
- If it moves by script or animations, make the Rigidbody kinematic. If it moves like a normal, physical object with forces and torque, don't make it kinematic.
- If you need scripts to respond to collisions between two GameObjects, at least one of those GameObjects needs to have a Rigidbody attached.

For our Player GameObject, we don't have a Rigidbody attached. This is because we're using a CharacterController to move our Player, and the CharacterController acts as a Rigidbody.

Layers are another concept that can influence collisions. Every GameObject is part of a single layer. You can see a GameObject's layer in the header of the Inspector, with the text "Layer" beside a dropdown field. The layer "Default" is automatically assigned to new GameObjects, so all of ours will have this layer. A GameObject has to be part of some layer. There are some set up in Unity by default, which you'll see if you click the Layer dropdown for any GameObject (see Figure 16-2).

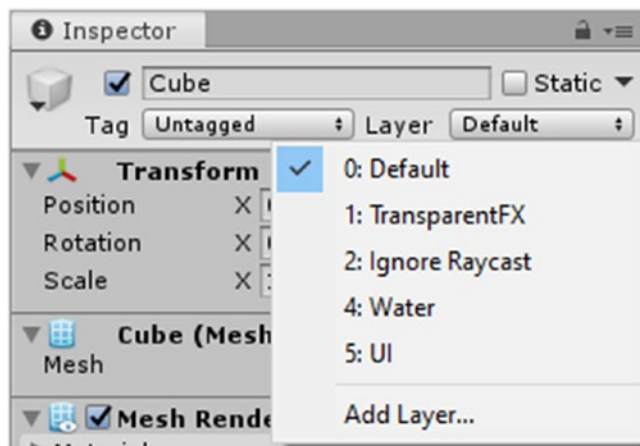


Figure 16-2. The Layer dropdown button for a generic Cube GameObject in the Inspector. The button has been clicked, which expands the list to show available layers. Clicking a layer will assign it to the GameObject

You can also create your own layers and name them whatever you like. To do this, go from the Layer dropdown menu of any GameObject and click the “Add Layer...” button at the bottom of the list (visible in Figure 16-2).

The Inspector will change to show a list of all layers, as pictured in Figure 16-3.

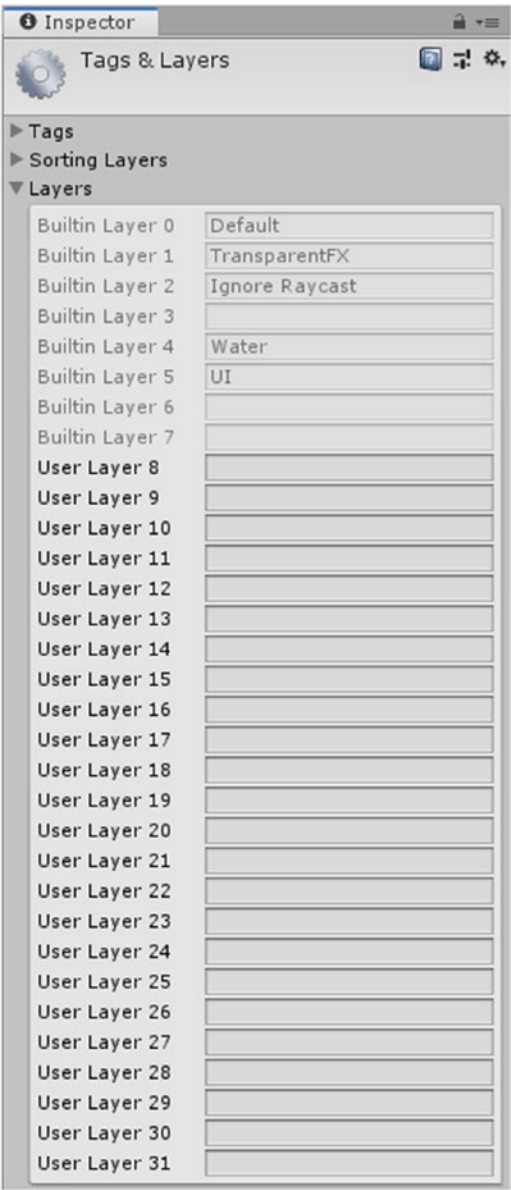


Figure 16-3. The Inspector “Tags & Layers” view, showing fields for each layer

There are a total of 32 slots you can use for layers. The first eight (layers 0–7) are built-in layers that you cannot rename. Layers 8–31 are all reserved for us, the game developer. We can type a name into any of these slots to make the layer available for assigning to a GameObject.

You’re probably wondering what the point of them is. The two major purposes are rendering and collisions. You can make cameras only render GameObjects of certain layers, and you can make certain layers only collide with certain other layers. For example, you could selectively hide GameObjects from the player camera or distinguish between friend and foe so that the player is not hurt by friendly attacks or projectiles.

Let’s declare layers for our project so that we can have fine-tuned control over which objects will be colliding with each other.

Name layer 8 “Player” and layer 9 “Hazard.”

Now select your Player to revert the Inspector out of the layer menu. Assign the Player layer to the player. Now that we’ve named it, it’ll show up in the dropdown list. When you change the layer of a GameObject that has children, Unity will ask if you want to change the layer of the children as well. That’s what we want, so select the “Yes, change children” option, and all of the children of our Player GameObject will be placed in the Player layer as well.

As of now, all layers will collide with all other layers. We can change that through the Edit ► Project Settings window. Click the Physics entry on the left side of your screen, and if necessary, scroll down to the bottom of the contents on the right side of the window. You’ll see something like a staircase of upside-down checkboxes, as shown in Figure 16-4.

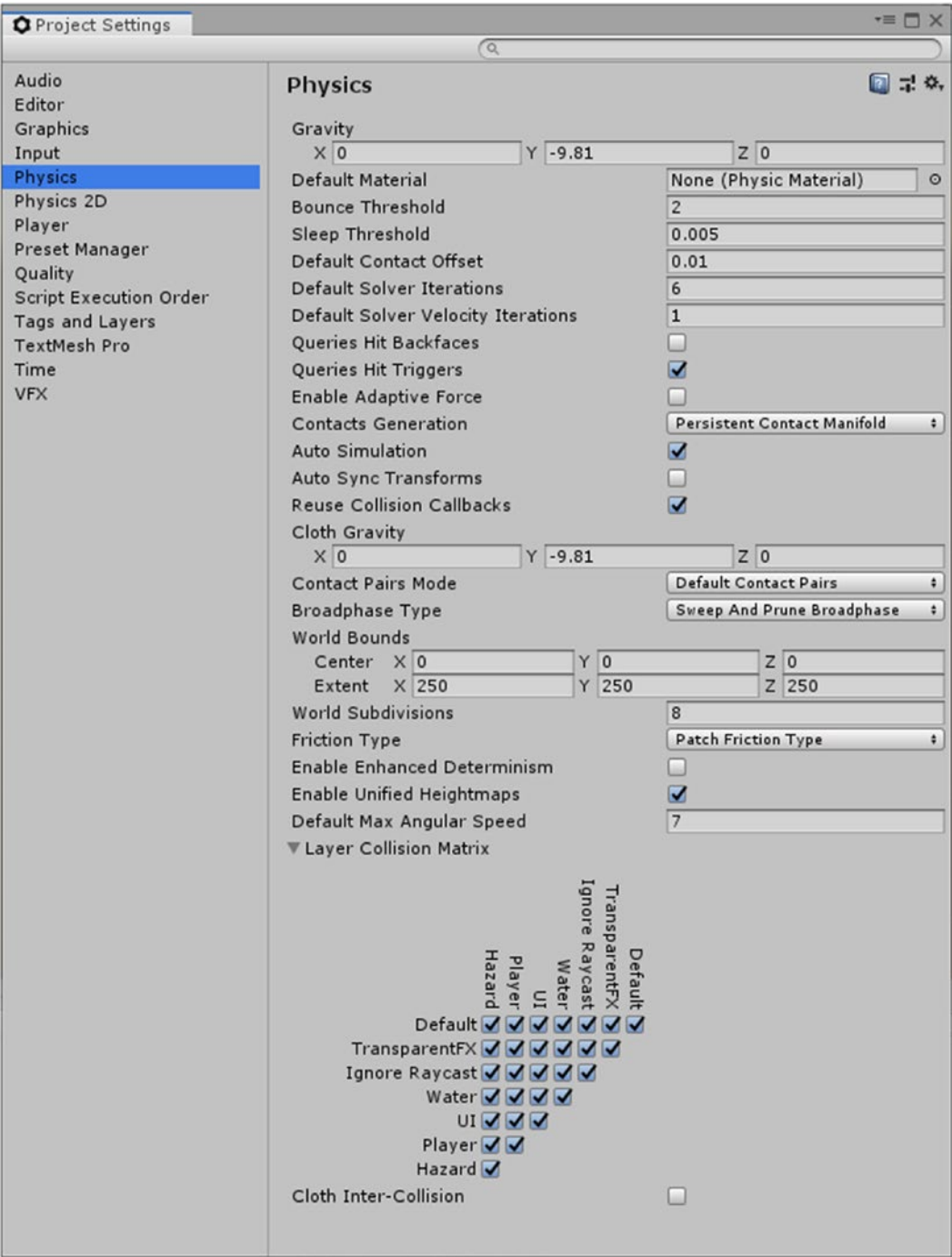


Figure 16-4. The Project Settings window, showing the Physics section. At the bottom, the Layer Collision Matrix field is unfolded to show an array of checkboxes corresponding to layer names

If you don't see it, you probably need to click the field titled "Layer Collision Matrix" at the bottom to unfold it so the checkboxes show.

This "collision matrix" depicts which layers are able to touch each other. There are layer names on the left side and the top side of all these checkboxes, and where those layer names cross, that is the layer combination that the checkbox is associated with. Easier yet, if you place your mouse over any of the checkboxes, it will display the two layers associated with it – for example, "Default/Player" or "Hazard/Player." If the box is checked, these layers can collide with each other. If it is unchecked, the layers will pass through each other. You can even make a layer unable to collide with objects of the same layer, if you want.

This can be useful to ensure that the hazards don't touch anything they don't need to touch. We only need them touching the player; we may also want them to hit walls and the floor. Generally, the Default layer can be used for environmental objects, so we'll plan to leave our floor and walls in the Default layer and enable collisions between Hazard and Player and Hazard and Default. As far as we're concerned, we don't want the Hazards colliding with anything else. We can even disable their collisions with themselves, so that traveling projectiles don't touch each other. To set it up like this, the matrix should look as pictured in Figure 16-5.

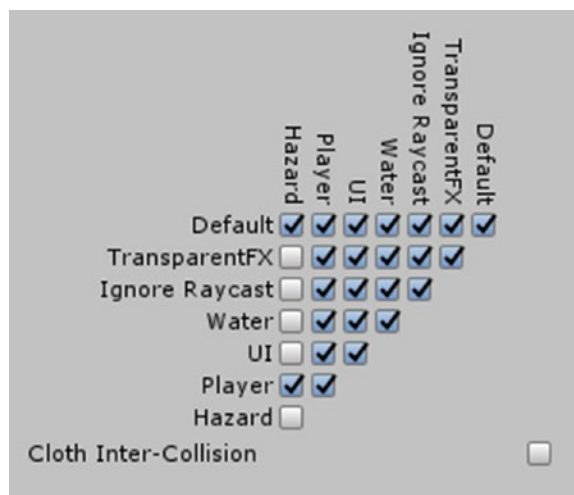


Figure 16-5. The Layer Collision Matrix field after we've set our Hazards to only collide with Default and Player layers

Hazard Script

Let's get to creating our Hazard script. This will make a `GameObject` kill the player on the spot when they touch. Create a script named Hazard in your project and open it up in your code editor.

The collision will be handled by Unity, and we'll set up our collider to make the shape we want our hazard to use. Our code just has to define what happens when the collision occurs.

There are built-in script event methods for this. The one we'll be using is `OnTriggerEnter`. It occurs once when a trigger collider of ours first hits another collider. It won't occur again unless the object stops touching our collider and then touches it again. It takes a single parameter, of type `Collider`, which will be given by the Unity engine when it calls the event. This parameter points to the other `Collider` that we touched, so you can name it "other."

Here's how the declaration will look:

```
private void OnTriggerEnter(Collider other)
{
}
```

Of course, nest it directly in the Hazard class.

Inside this method, we can use the "other" parameter to grab the associated `gameObject` (the one we hit); and through the `gameObject`, we can check the layer. This is an int variable that depicts the number of the layer. Remember, 0 is Default, 8 is our Player layer, and 9 is our Hazard layer.

We'll check if the layer is 8, to make sure we've hit the player. We may have already set up our physics detection such that Hazards only touch the Player, but you never know what might happen down the road, so we'll check to make sure anyway:

```
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.layer == 8)
    {
        Player player = other.GetComponent<Player>();
```

```

        if (player != null)
            player.Die();
    }
}

```

Inside that if, we create a local variable and attempt to get the Player script component using this new GetComponent method.

The GetComponent method can be accessed through any other Component reference. In this case, we use the Collider “other.” It can also be grabbed through a GameObject reference.

The <...> part is a new concept for us. To put it loosely, it is the concept of **generics** – a feature of C# that allows types to be passed around as parameters to classes and methods, among other things, by listing the types between angle brackets (the < and > symbols).

The GetComponent method declares a **generic type parameter** that specifies which type of component we want to get. Within the GetComponent method, this parameter can be referenced as though it was an actual type, allowing some pretty cool and powerful behavior to be defined. Rather than returning some specific type, the GetComponent method returns the generic type it is given.

This means the return type effectively changes on the spot to whatever type we pass in when we call the method. You can’t do that with a normal parameter. This allows us to call GetComponent<Player> and receive a Player back instead of a Component. This way, we don’t have to make a typecast to apply the reference to our new local variable. If we didn’t do this, we’d have to specify the type of component through a normal parameter, and we’d get a Component returned to us instead. As a less specific class, the Component instance can’t be stored in a Player reference, and we’d get an error. We’d have to fix it with a typecast. Ultimately, it looks much cleaner to use the generic version of the method.

Generics are a somewhat complicated topic, so I could definitely say more about them here, but luckily, you don’t need to be well-versed in the inner workings of generics to simply use them in method calls. That’s a path you can go down at a later time.

Anyway, after we call the GetComponent method, we then check to see if we actually got anything from it. If the GameObject didn’t have a Player component on it, the GetComponent method would simply return null. We only want to proceed if that did not happen (player != null), in which case we call our method player.Die().

That does it for the Hazard script. Let's test it real quick by creating a stationary Hazard that we can touch with the player:

- Create a Sphere. Set its scale to (3, 3, 3). Place it somewhere near the player, either by dragging it with the position handle (hotkey W) or copying the player X and Z positions into the Sphere position in the Inspector. Set its Y position to 1.5 to make sure it's resting on the floor neatly.
- Set the Sphere's layer to Hazard in the Inspector.
- Check the "Is Trigger" field of the Sphere Collider component in the Inspector. It won't receive OnTriggerEnter calls if it isn't marked as a trigger collider.
- Attach a Hazard script component to the sphere.

Now play the game and run your Player into the Sphere. It should kill the player on contact.

You can delete the Sphere after you've finished playing around with it.

Projectile Script

Now we can create a Projectile script that provides forward movement. When we get around to it, we'll make our Shooting script, which will spawn the projectiles and point their forward axis in the direction they should be traveling. Knowing that, all we need to do is make the projectile move forward until it reaches its maximum range.

We'll create a Projectile script and declare some variables:

```
[Header("References")]
public Transform trans;

[Header("Stats")]
[Tooltip("How many units the projectile will move forward per second.")]
public float speed = 34;

[Tooltip("The distance the projectile will travel before it comes to a stop.")]
public float range = 70;

private Vector3 spawnPoint;
```

You’ve read the tooltips, so you ought to know what the variables mean. We use the same “trans” reference, since it should be a little bit faster than referencing “transform” itself. The spawnPoint will be set **in Start**, to memorize where the projectile was when it spawned:

```
void Start()
{
    spawnPoint = trans.position;
}
```

We’ll use that to make sure it doesn’t travel further than it should, **in the Update method**:

```
void Update()
{
    //Move the projectile along its local Z axis (forward):
    trans.Translate(0,0,speed * Time.deltaTime,Space.Self);

    //Destroy the projectile if it has traveled to or past its range:
    if (Vector3.Distance(trans.position,spawnPoint) >= range)
        Destroy(gameObject);
}
```

This time, we use the method “Translate” to move the Transform. This method has various overloads that allow slightly different parameter inputs, but the one we’ve chosen is using three float values for the X, Y, and Z movement we want and an enum “Space,” which determines whether the Transform should be moved in world or local coordinates. This enum only has two options: Space.World and Space.Self (“self” meaning “local”).

This parameter has a default value of Space.World. This means if you want, you can omit the parameter entirely, and it will assume its default value and run just fine. However, since we want to rotate the projectile however we like and have it always go along its own forward axis, that means we want local movement instead, so we need to specify the parameter.

After applying the movement, we use another new method, Vector3.Distance. This takes two Vector3 instances and returns the distance between them as a float. We check if the distance between the projectile and the spawn point has exceeded the “range”

variable. If so, we destroy the `GameObject` that the `Projectile` script is attached to, with a method `Destroy`, available through any Unity object.

Now let's create a projectile prefab. We'll set it up in the scene and then drag and drop it to the Project to make the prefab asset.

It'll be pretty simple:

- Create a `Sphere` named `Projectile`. Set its layer to `Hazard`.
- Check the `Is Trigger` box in its collider.
- Add the `Hazard` and `Projectile` scripts. Set the "trans" reference for the `Projectile` script by dragging and dropping the `Transform` component right there in the Inspector.
- Add a `Rigidbody`. Check the `Is Kinematic` box. You can uncheck the `Use Gravity` box if you want, but gravity won't apply to a kinematic `Rigidbody` anyway.
- Create a new `Material` in your Project view. Name it `Projectile`. I've picked a slightly off-red color for mine, with a hex value of `E81010`.
- Drag the new material from the Project window onto the sphere in the Scene window.
- Now we're done, so make a prefab for the projectile by dragging it from the Hierarchy to the Assets ► Prefabs folder in the Project. If you never made the folder, you can do so now. Or if you're that type, you can just be messy and throw the prefab anywhere in there.

If everything is set up correctly, your `Projectile` should look like Figure 16-6 in the Inspector.

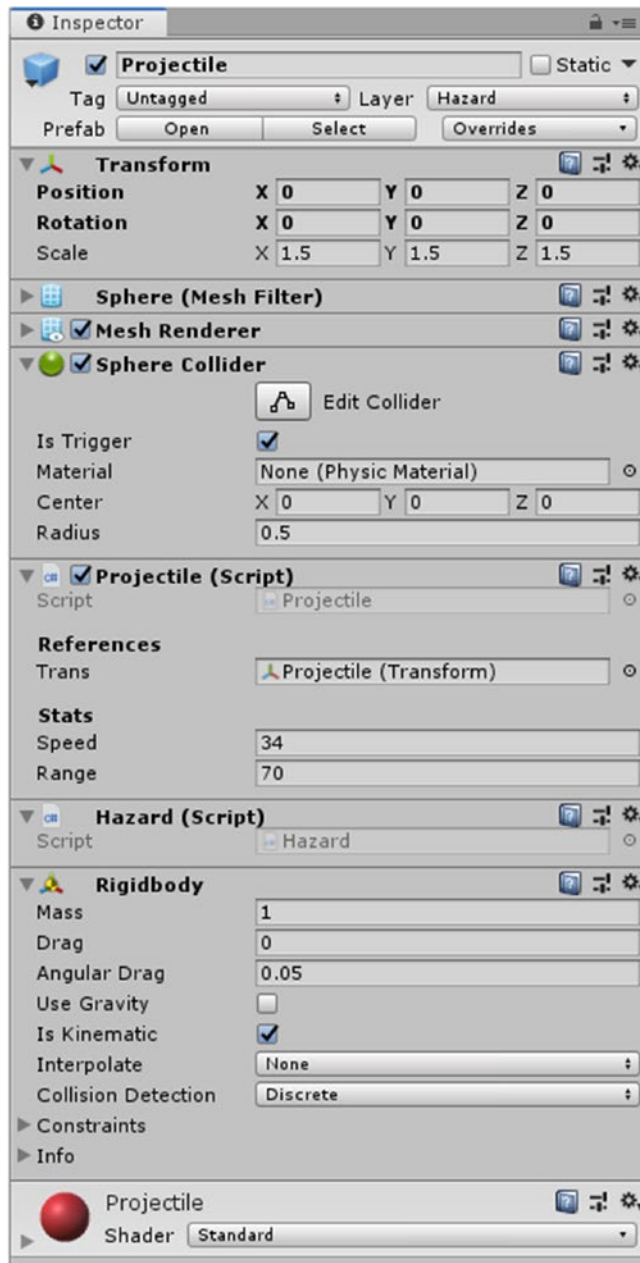


Figure 16-6. The Inspector view of our fully setup Projectile

If you want, you can give it a test and position the projectile somewhere in your scene, play the game, and watch it move. You can even jump in front of it with the player to see the Hazard script in action again, if you're so inclined.

Once you're done playing with it, you can safely remove the projectile instance from the scene, since we've made a prefab of it already.

Shooting Script

Now we'll create a "shooter" obstacle and a Shooting script to fire projectiles. This will give us our first example of spawning GameObjects on the fly through code.

Let's build the GameObject. We're going to use a very minimal amount of creativity to make this one. We'll use a cube for its base and a cylinder "barrel" poking out of it. It should look something like Figure 16-7 by the time you're done with it.

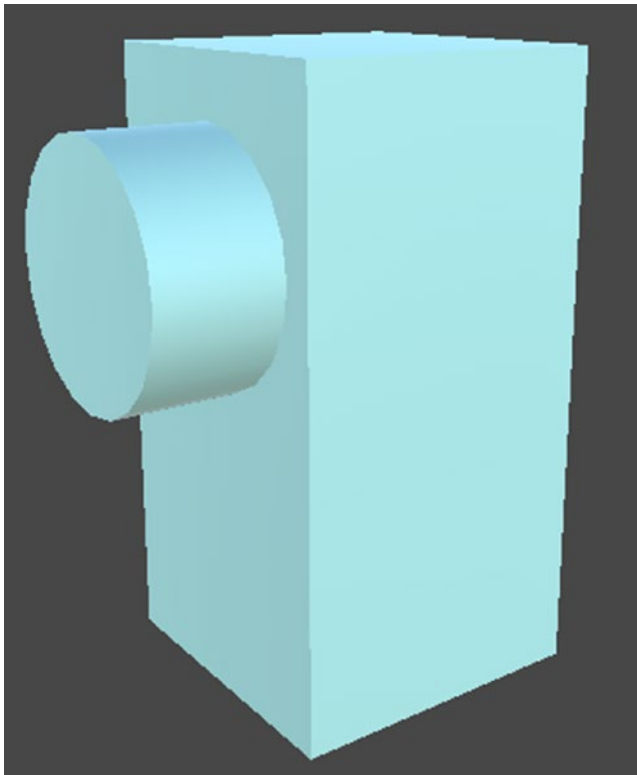


Figure 16-7. *The final appearance of our Shooter GameObject*

Here's how you'll go about putting it together:

- Create an empty GameObject and name it Shooter.
- Create a cube as a child of Shooter. Name it Base. Set its local position to (0, 2, 0) and its local scale to (2, 4, 2). You can leave the collider on it as is, so that if we ever bump into the shooter itself, we'll slide against it like a wall.
- Create a cylinder as a child of Base. Name it Barrel. Give it a local position of (0, .2, .7), a local rotation of (90, 0, 0), and a local scale of (.8, .2, .4). It should now be a short cylinder poking out the front side of the Base cube, near to the top.
- Create an empty GameObject with no initial parent (Ctrl+Shift+N). Name it Spawn Point. Drag it onto the Barrel in the Hierarchy to make it a child. Leave the local scale and rotation as is. Set its local position to (0, 1.5, 0) to put it a little bit in front of the barrel.
- If you want, make a Material named Shooter and apply it to your Base and Barrel. I've colored mine a pale cyan with a hex value of A1DED1.

The Spawn Point will be an invisible GameObject used for its position and rotation. We'll spawn new projectiles there and rotate them to point in the same direction.

Let's get to it. Make a Shooting script and open it up.

This one's pretty short, and you're an expert at this process now, so here it goes:

```
[Header("References")]
public Transform spawnPoint;
public GameObject projectilePrefab;

[Header("Stats")]
[Tooltip("Time, in seconds, between the firing of each projectile.")]
public float fireRate = 1;

private float lastFireTime = 0;
```

```

void Update()
{
    if (Time.time >= lastFireTime + fireRate)
    {
        lastFireTime = Time.time;
        Instantiate(projectilePrefab,spawnPoint.position,spawnPoint.
            rotation);
    }
}

```

Of course, those will be the contents inside the Shooting class. We have a reference to the Spawn Point Transform and to the GameObject of the Projectile prefab. We set these in the Inspector, as always.

We also have a public “fireRate” variable and a float which stores the current game time at which a projectile was last fired. To implement the wait time between each spawn, we use the Time.time variable, which gives the time, in seconds, since the game began. It starts at 0 and counts up over the course of the game being played. To see if it’s time to spawn a new projectile yet, we just need to check if the current time (Time.time) is greater than the Time.time at which we last fired (lastFireTime), plus the seconds between each firing (rateOfFire). Of course, in order for this to make any sense, we have to also reset the lastFireTime to the current Time.time again every time we fire a projectile.

To spawn a prefab on the fly, we use the built-in method Instantiate. It takes three parameters: a pointer to the thing we want to create, a Vector3 for the initial position we want it to have, and a Quaternion for the initial rotation we want it to have. We’ll point to the projectile prefab and use the position and rotation of the Spawn Point to neatly align the projectiles with the barrel.

That’s it! We can save the script and attach it to the base Shooter GameObject. Set the reference to Spawn Point by dragging it from the Hierarchy to the field in the Inspector. The projectile prefab can be dragged from the Project onto its respective field.

Create a prefab for the Shooter to make sure we don’t accidentally lose it somehow (having to rebuild it would be a little frustrating).

Now you can play and test the results. If you’ve set the Shooter up as described, it will keep spawning projectiles at the Spawn Point position, firing them off along its forward direction, toward where the so-called barrel points. Watching from your Scene window, you can travel out with the projectiles to see that they’re disappearing once they reach the end of their range. If your Shooter is not positioned at the correct vertical location,

just set the root “Shooter” GameObject Y position to 0, or delete it and drag and drop a new prefab instance down, which should snap to the correct Y location automatically since we made the pivot point at the bottom.

Summary

This chapter implemented our first obstacle type, a stationary Shooter that fires projectiles periodically. Some things to remember are as follows:

- A Rigidbody can be added to a GameObject to provide realistic physics simulation for that GameObject, including gravity and forces that push, twist, and turn the GameObject.
- A Rigidbody that is marked as kinematic will not be controlled by the physics system. If you’re going to directly move a Transform, its Rigidbody should be kinematic; otherwise, your influences on the Transform position will cause unpredictable disruptions to the natural physics performed by the Unity engine.
- For collision-related events to be sent on scripts, one of the colliding entities must have a Rigidbody attached, even if it’s a kinematic one.
- The CharacterController counts as a kinematic Rigidbody. You don’t need a Rigidbody attached to a GameObject that already has a CharacterController.
- Every GameObject is part of one layer. Layers can be used to determine which objects collide with each other and which do not.
- The OnTriggerEnter built-in event is called when a trigger collider first detects a collision with another collider.
- The Transform.Translate method is used to move a Transform by a given X, Y, and Z amount. The movement can be performed in world or local space using the fourth parameter.
- The Destroy method is available from within scripts and can be used to destroy a GameObject, removing it from the game world entirely.
- The Instantiate method is also available from within scripts. It can be used to create a new instance of a prefab, with the desired initial position and rotation as parameters.

CHAPTER 17

Walls and Goals

Now that we have some obstacles taking shape and our player movement working nicely, we can start thinking about setting up levels and a win condition for them: touching a goal object.

To block out our levels, we'll use simple cubes based off a prefab, with variations on their scale and position to stretch them out however we need to make our level. It might look a bit shoddy, but again, we're programmers, not designers.

Walls

Create a cube and name it Wall. Make a material, also named Wall, apply whatever color you like, and throw it on there. I'm using a brown with a hex value of 601E1E. We'll leave the collider as is. It won't be a trigger, because we want it to be physical and block the player. It doesn't need a Rigidbody because it doesn't move.

Make it as tall as you like and position it so the bottom is touching the floor. When we set up our "Floor" plane at the start of the project, we made sure to position it at (0, 0, 0). This means the floor has a Y position of 0, so you can perfectly touch the bottom of the wall to the floor by setting the wall Y position to half the Y scale. I have my wall at a size of 10 (which means a Y position of 5).

If you've accidentally moved your Floor plane, you can fix it by setting its Y position to 0 again, although you'll have to adjust any other GameObjects in the scene to match.

Now we can make a prefab out of this wall. Going forward, we'll make sure all the walls we add to any of our levels are instances of this prefab. Yes, it was easy to make, but if we make new ones each time or make a separate one for each level and copy-paste it, we won't be able to make a change to all of them at once if we should ever need to.

To shape a level and confine the player within it, we can copy-paste instances of this prefab and change their position, rotation, and scale however we like, letting them overlap and stick together.

Whenever we start a new level (we'll make each one in a separate scene), we just throw a wall prefab instance down, reposition its Y correctly (its pivot point is at the center, so it won't line up to the floor correctly when we place it), and then just copy-paste that to make more instances on a dime.

The rect transform tool (hotkey T) can be particularly useful in quickly sizing a block like this. Just make sure you set the Tool Handle Rotation control (hotkey X) to Local, not Global, so that when working with rotated walls, the rect tool matches the rotation of the selected wall. Figure 17-1 shows the location of the Tool Handle Rotation button up at the right side of the transform tools in the toolbar.



Figure 17-1. The Tool Handle Rotation, highlighted with a red box, is shown in the Local setting

You can then drag the edges or corners of the walls to pull them, allowing you to easily stretch and shrink the walls.

I'd recommend never changing the Y scale of individual walls, to keep them all uniformly flat on top. It also means the Y scale remains original, so we can change it for all our walls at once with a change to the prefab, although the pivot point complicates this because their Y position will have to be shifted down to make them touch the floor again (otherwise, they'll hover above it if you made them shorter or stick through it if you made them taller).

Another little trick you might be interested in when setting up walls this way would be to get a nice overhead view of the scene. That little gizmo in the top-right corner of the Scene window comes in handy here (see Figure 17-2).



Figure 17-2. The gizmo displayed in the top-right corner of the Scene window

You can click the gray cube at the center of the gizmo to switch between Perspective and Isometric camera modes. The current setting is displayed in text beneath the gizmo (Persp or Iso). Just click that cube to toggle between the two settings. Isometric mode makes everything look “more 2D,” so to speak, while Perspective mode looks more standard, with field of view stretching objects as they near the edges of the screen. Isometric mode can make it easier to line objects up accurately.

The gizmo also shows little cones pointing off the central cube. Three of these cones are color-coded by their axis (red, green, and blue) and labeled for the X, Y, and Z axes. Any of the cones can be clicked to spin the camera around to view the world from straight on at that angle: top, bottom, left, right, and so on, depending upon which cone you clicked. If you click the green cone, you get a view from the top of the world, letting you get a bird’s-eye view of the level as you edit it.

As we develop more functionality to our game, you can play with your own level designs and get creative with it.

Goals

Alright, let’s make a simple Goal script to define a means of winning the level. This is going to be much like a Hazard: detect a touch from the Player and run some code when it happens.

Let’s make a layer to use for our goals first. Using the Layer dropdown found in the head of the Inspector for any GameObject, click “Add Layer...” to reach the layer menu just as we did before, and write the name “Goal” in the User Layer 10 slot.

Once again, go to the collision detection matrix by navigating to Edit ► Project Settings, selecting the Physics tab on the left, and then scrolling down to the Layer Collision Matrix field at the bottom. Make the Goal only collide with the Player – nothing else, shown in Figure 17-3.

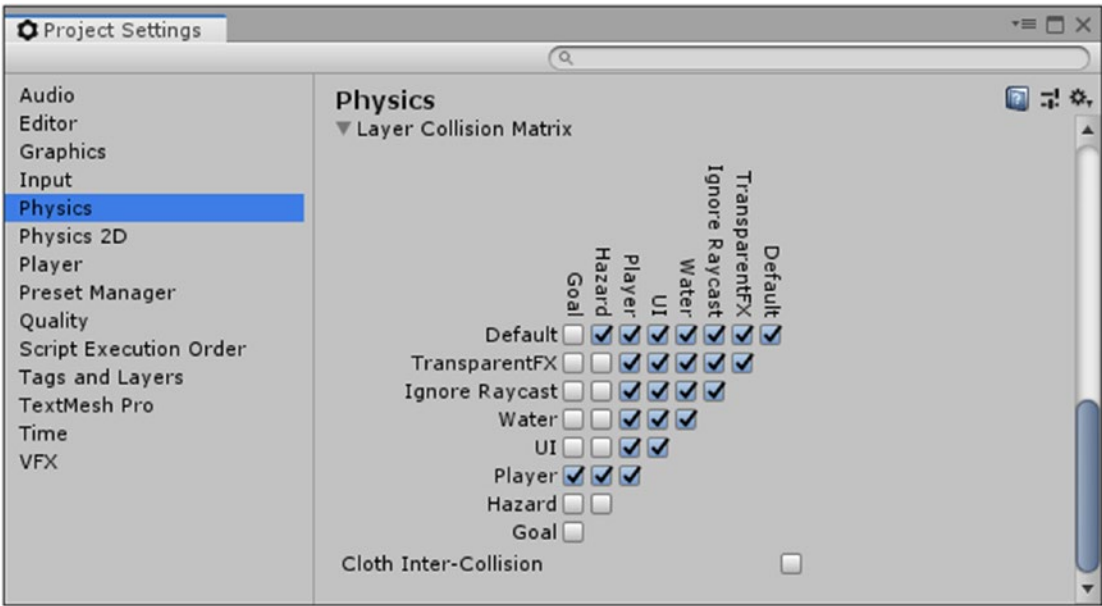


Figure 17-3. Within the Physics tab of the Project Settings window, we have scrolled down to the Layer Collision Matrix and set our Goal layer to collide only with the Player layer

Let’s create the GameObject for it while we’re at it:

- Create an empty GameObject named Goal.
- Add a cylinder child and scale it to (4, .1, 4), making it like a thin disc. Give it a (0, .1, 0) local Y position. Check the “Is Trigger” field in its collider.
- If you want, make a material “Goal,” apply it to the cylinder, and give it some color. I’m using a bright green (the color of success in video games), hex value 2CFF28.
- Apply the Goal layer to the root GameObject (the one aptly named Goal). This should also provide an option to apply the layer to all children as well. Say “Yes, change children” to that.
- Give the root Goal GameObject a kinematic Rigidbody. There may not be a collider attached to the Goal itself, but the Rigidbody will use the child cylinder’s collider without any further setup on our part.

By the end, it should look something like Figure 17-4. The collider sticks up past the cylinder a little, but that won't hurt anything.

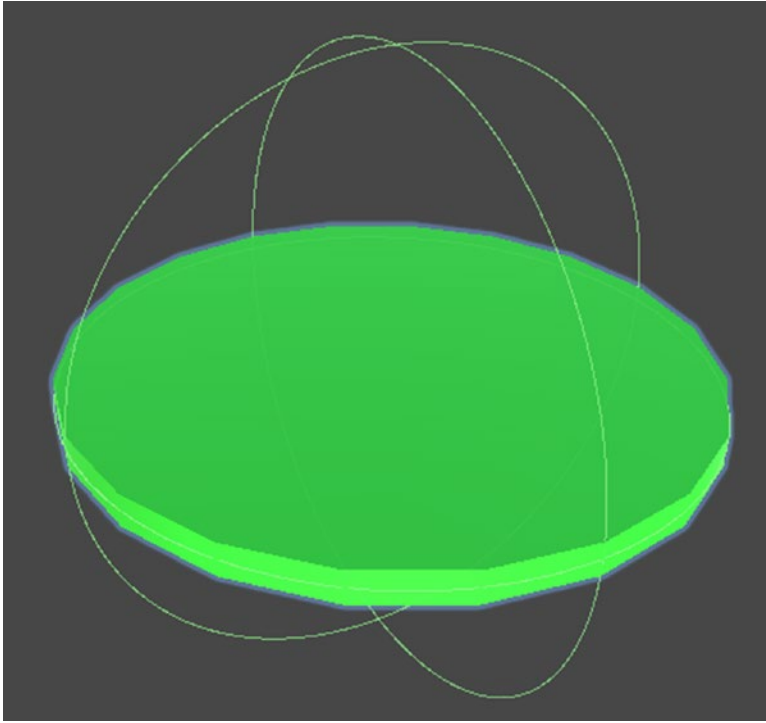


Figure 17-4. Our Goal GameObject, with the shape of its collider visible as green lines arcing over and beneath it

Now, create a Goal script in your Scripts folder with the Project window. Attach an instance of the new script to the Goal root GameObject.

While we're at it, make a prefab out of the Goal by dragging it to the Prefabs folder in your Project window.

The Goal script will be pretty short: when the player touches the Goal, we'll load the "main" scene again. We haven't gotten there yet, but eventually, each level is going to be in its own scene, and the "main" scene will be the menu that the user sees when they start the game, allowing them to select a level to play on. For now, we'll just send them over there again when they win. It's not much of a rewarding experience, but it's the journey that counts, right?

To do this, we need to add a new “using” statement at the top of our script file, up there with the rest of them. You’ve probably paid little attention to these since their first mention. To recap, they exist to tell the compiler what other “namespaces” (tidy containers for related code definitions) we want to use in this script file.

We need a particular namespace that allows us to run a method to change the current scene in-game. Add this “using” anywhere in there with the rest, up **at the top of the script file**:

```
using UnityEngine.SceneManagement;
```

Now add this code **within the script class**:

```
void OnTriggerEnter(Collider other)
{
    if (other.gameObject.layer == 8)
        SceneManager.LoadScene("main");
}
```

We’ve seen most of this before: we detect a touch from our trigger collider and make sure the other GameObject is the player (layer 8), and if so, we run this method to load a scene by its name, “main”. If you never actually named the scene “main”, either do that now or just write whatever name your scene uses in the method call (but make sure you don’t mistype it).

This method can be customized with an extra parameter to specify that we want to load the scene “additively,” which means the loaded scene gets mixed in with the current one and none of the GameObjects of our current scene get lost. But if you don’t specify this, it loads the new scene in by replacing the existing scene entirely (including the Player). Since we’re just loading the same scene again, it will reset everything back to its original state. In this case, that’s what we want.

We’re not done just yet, though. Before we can load a scene in-game, that scene has to be added to our Build Settings. Let’s learn what that means.

Build Settings for Scenes

If we ever want to turn our game into a program that we can send to users to play without use of the Unity editor, we need to **build** the project. This copies our Unity project into a set of files that can run the game independently of the editor – for example, as

an executable (.exe) file. But first we need to add any scenes we're using to the Build Settings. If the scenes aren't in the Build Settings, they don't get "put in the build" and can't be loaded in-game. We may be able to open them and play them in the Unity editor, but if we want to load them through a script, they must be added to the build.

The Build Settings (see Figure 17-5) are accessed by heading to File ► Build Settings or with the hotkey Ctrl+Shift+B.

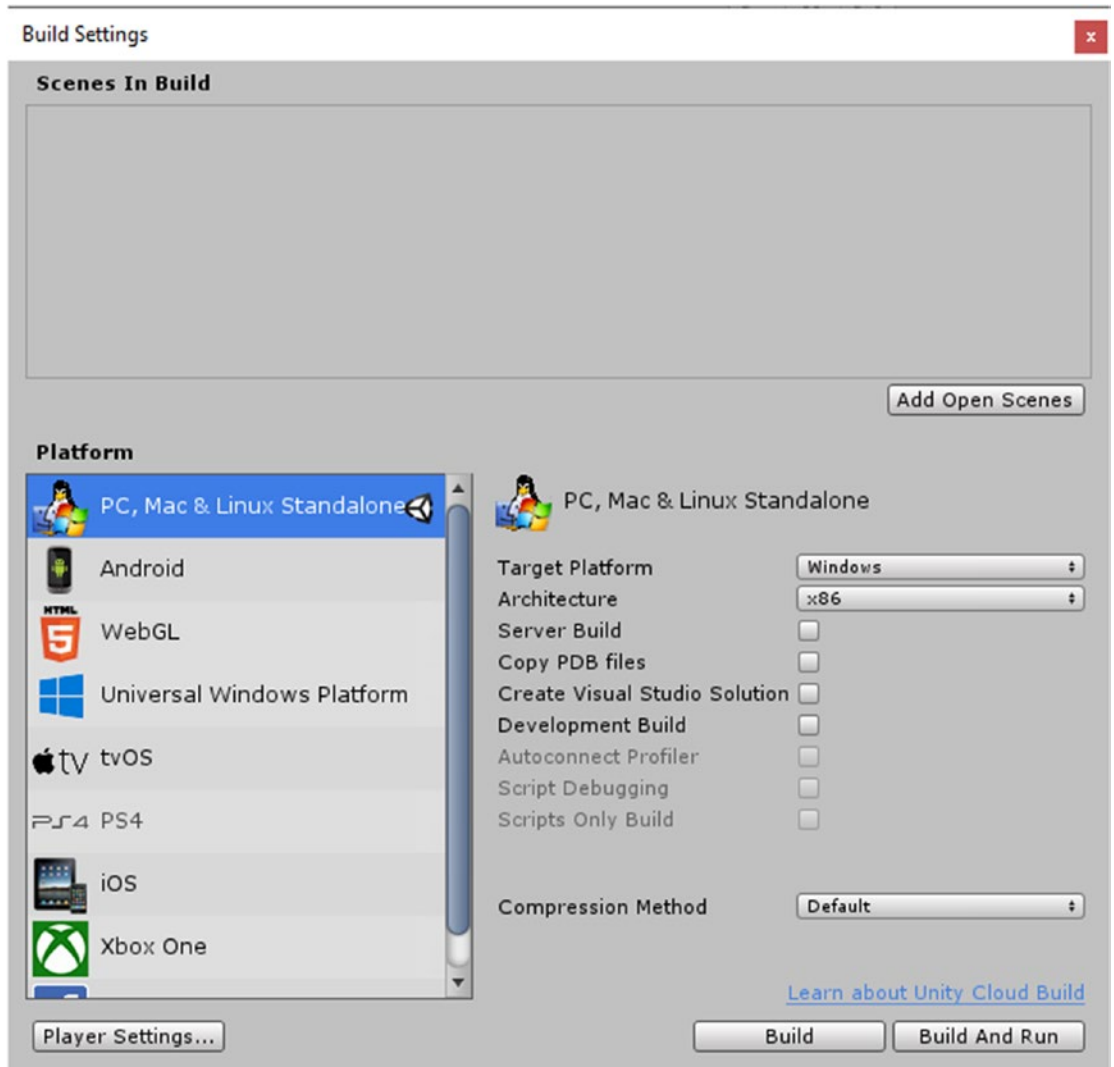


Figure 17-5. The Build Settings window

This is where you can find your target platform, which should be “PC, Mac & Linux Standalone” by default. On the right, we have relevant options specifying how the project would be built, such as the target platform (Windows, Mac, or Linux, although you’ll need to have installed the appropriate build modules in the Unity Hub to see all these options). To the left, there’s a listing of other platforms available to target. You can switch the target platform here. That’s how you would target things like WebGL (to let your game run in a web browser) or even game consoles. We won’t dive into those topics right now, though.

We’re focused on the top portion of the window, the “Scenes In Build” section. Right now, it’s just an empty rectangle. At the bottom-right corner, we can click the button “Add Open Scenes” to put our currently opened “main” scene into that list, as pictured in Figure 17-6.

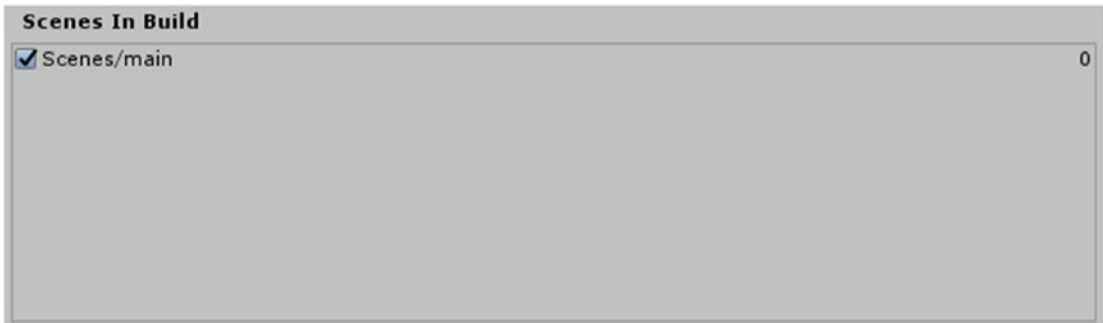


Figure 17-6. The *Scenes In Build* section of the *Build Settings* window, after we have added our main scene to the list

This will include the scene in the game when the game is built. We only need to do it this one time, and the scene will remain in that list. If we want to delete a scene, we can select it and press Delete on our keyboard or right-click and click Remove Selection.

All the way at the right side of the entry for our “main” scene, you’ll see a little number 0. This is the scene **build index**. It starts at 0 and goes up as we add scenes. If we had other scenes in here, we could reorder them by dragging and dropping with the left mouse button.

The build index is another means of loading a scene with the method we used in our Goal script. Now that our “main” scene is in here, we could go back to our Goal script and change the “main” string to use the build index instead, meaning we’d just pass a 0 as the parameter. In our case, it’ll do the same thing.

But the most important thing about adding the scene to the build settings is to ensure that it actually makes it into the build and to designate it as the first scene in the project, so that it's the first thing that loads when you start the game. In the Unity editor, we just load whichever scene we want and don't worry about this, but if we were to make a finished product and send it to players, we'd need to send them to the correct scene when they first load the game up. Similarly, any level scenes we want to load for the player will have to be added to the build settings to make them available when playing a built version of the game.

The scene at build index 0 is always the one that will load first when the game is run from a build. Since our "main" scene is intended to be the home screen where the user selects the level they wish to play, we want that to be our first scene.

Now that the main scene is part of the build settings, the goal should function just as expected. With a goal placed down somewhere the player can touch it, go ahead and test it out. Touching the goal should reset the scene back to its original state, reloading all the GameObjects within. It might come with a little bit of a hiccup. That's expected – loading in new scenes can come with a short wait.

Summary

This chapter taught us how to set up walls for our level and how to edit them conveniently with the rect tool. We've coded our Goal script and set up a prefab for a consistent Goal GameObject which will take the player back to the "main" scene when they touch the goal. Some points to remember are as follows:

- To access the class required to load a scene, include a "**using UnityEngine.SceneManagement;**" line at the very top of the script file.
- If you want to load a scene in-game, it must be added to the build in the Build Settings window.
- In the list of scenes added to the build, each scene is assigned its own **build index**, a number based on the scene's order in the list.
- To perform a scene load, call the **SceneManager.LoadScene** method. You can use a string for the name of the scene or an int for the scene build index.

CHAPTER 18

Patrolling Hazards

Now that the game is shaping up more, it's in need of some greater obstacle variety. In this chapter, we'll implement "patrollers."

Patrollers are objects that we set up with a series of points they'll travel along. They move from one point to the next, starting at the first point and going down the list. Once they reach the last point, they double back to the first point again and then repeat the process. We can set the points up however we want and use however many points we want. If we want an obstacle to travel in a simple line back and forth, we can just use two points: one at one end of the line and another at the opposite end. Alternatively, we could have them run along a series of points that lead them down a more complicated path.

Resembling a Patrol Point

The first challenge we'll need to face is managing the patrol points. How are we going to resemble each patrol point and access it in our Patroller script?

The only data we really need for patrol points is a position – that's all they really are, right? So we could expose `Vector3` instances in the Inspector and manually type in the position of each patrol point. But that's not very convenient.

Instead, we can use empty `GameObjects` in the Scene to resemble patrol points. This way, we can position them using the transform tools and view them in the Scene to know where they actually are. Then, we can set references to each patrol point Transform and grab its ".position".

If we'd used `Vector3`s, we wouldn't really be able to visualize where a point is. We'd have to make a `GameObject`, put it where we want, and then copy its position value into a field in our Patroller script. And if we're doing that, why not just use empty `GameObjects` as the points instead?

The next challenge we face is how we'd go about referencing all of these patrol point Transforms in our Patroller script. We might think to declare a bunch of variables, one for each patrol point. We'd name them `patrolPoint1`, `patrolPoint2`, and so on. We'd set them as references to patrol point Transforms in the Inspector. We could declare, say, ten of them, because we don't think we'll ever need more than ten. Any points that aren't set (meaning the value is null) would be ignored, so we'd double back to point 1 as soon as we reach the last point that isn't null.

But that's kind of awful. Not only is it a pain to code the logic for switching from one point to the next, it's simply not doing all we want it to do. It's limiting us to however many points we're willing to spend the time creating variables for, and it requires that we set a reference to each point.

This is where **arrays** come into play.

Arrays

Arrays are a means of storing a collection of a certain data type in a single object. An instance of an array is given a type (such as `Transform`) and a length. The length is the number of items that the array stores.

Say we create an array of type `Transform`, with a length of 20. We now have a single object that stores 20 separate `Transform` references, all of which are initially set to null.

But how do we access individual items within the array?

This is known as **indexing**. Each item in the array corresponds to an index – an integer value, like an ID that can be used to point to a specific “slot” in the array. The index of the first item will be 0. The next item will be 1 and then the next 2, going all of the way up to 19 in an array with a length of 20 (since it starts at 0, not 1, the last index will always be 1 point less than the length).

To declare an array, you declare it like a normal variable, but after the type, you put an empty set of square braces `[]`:

```
public Transform[] arrayOfTransforms;
public Vector3[] arrayOfPoints;
```

To create an instance of an array, it's much like calling a constructor to create an instance of a class as you normally would. The difference is that we use square braces `[]` instead of parentheses `()` after the type name. Within those square braces, we put a single

parameter, which is the number of entries the array will store. This parameter resembles the array length. It can be accessed at any time through the `.Length` member:

```
//Create an array storing 5 Transforms:
arrayOfTransforms = new Transform[5];

//Log the length:
Debug.Log(arrayOfTransform.Length); //This will log the number 5
```

In the preceding code, we've declared an array storing up to five Transforms. The first has an index of 0, and the rest have indexes of 1, 2, 3, and 4. Notice that the last index is equal to `Length - 1`, not `Length`. That's due to the index starting at 0 instead of 1.

To access the Transforms through the array, we reference the array and then type an **indexer**, which is a square brace set `[]` with an integer value inside it. That integer value is the index of the item we want to grab out of the array. For example, we would grab the first Transform in the array like this:

```
Transform first = arrayOfTransforms[0];
```

Often when working with arrays, we don't just pass in literal values like 0 or 4 as our index. It kind of eliminates the purpose of arrays.

What we can do is keep an integer "currentPointIndex" variable in our Patroller and use that to keep track of the index of the point we're currently traveling toward.

When we reach the point, we increase `currentPointIndex` by 1 and then use it as an index to grab the next point in the array.

If you ever try to grab from an array using an index that's too high or is a negative value, an error will be thrown. It doesn't just double back to the beginning automatically for us, so we have to account for that ourselves.

Luckily, that's not so hard. Whenever we're ready to switch to the next point, we simply check if the `currentPointIndex` is already the last index in the array, which is equal to `array.Length - 1`. If so, we set the `currentPointIndex` back to 0. If not, we just increase it by 1.

This creates a neat loop through the items of the array, and it works no matter how many items we store in the array.

Setting Up Patrol Points

Let's figure out how we plan on setting up references to our patrol points before we begin coding anything.

Arrays can be serialized just like other data types (so long as they store a type that also supports serialization). If you'll recall, a value being serialized means it shows up in the Inspector so that we can edit it and save its state within the Scene.

When an array is serialized, you can set its length to whatever you want, and Unity will give you a field for each item (index) in the array. We could then drag and drop the Transforms of our patrol points into those fields to set the array up manually. It would look something like this in the Inspector, shown in Figure 18-1.

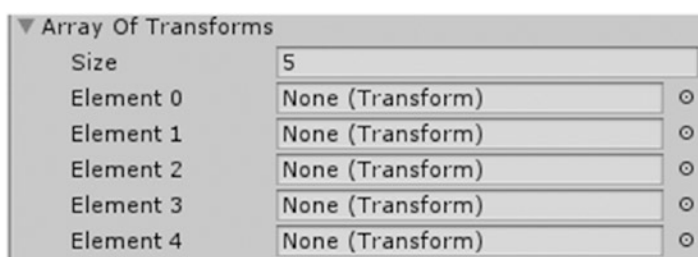


Figure 18-1. A view of a serialized array of Transforms in the Inspector. We've set the length to 5 manually. None of the references have been set yet, showing as "None" (equivalent to null)

But we're not chumps, so we won't be doing that. Dragging and dropping patrol points into this array to set up the patroller will be somewhat tedious, so we'll be setting the references through code instead.

We want to automate this part of our workflow so it's easier for us to set up patrollers. It may be slightly less performant as a result. Finding the patrol points at runtime is always going to be slower than just setting up the references directly in the Scene, when the game isn't playing. But the performance hit shouldn't be noticeable anyway, particularly for a game with such a small scope. Sometimes it's worth it to implement a simple feature like this to make the game easier to develop. If you're too hard on yourself just to make the game run as fast as possible, you might end up never releasing the game in the first place because developing it drove you crazy!

So what do we have to do to set up these patrol points and grab the references to them in our code?

In order to find all of the patrol points associated with a single Patroller instance through code, we'll be grabbing all of the children of the GameObject with the Patroller script attached. So all of the empty GameObjects we use to resemble patrol points will have to be children of the Patroller they belong to. This keeps them neatly tucked inside the Patroller they belong to in the Hierarchy, so it's a win-win.

But we don't want them to move or spin with the Patroller, so we'll just unparent them as soon as we find them and set up our references. That way, they stay put in the scene even as the Patroller moves and rotates. It would be anarchy if we forgot this step. The Patroller would keep moving toward a point that stays a fixed distance away from it, so it'd never reach the first point at all.

We need to distinguish patrol points from other children that aren't meant to be patrol points. We're simply going to use the GameObject name to do that. We'll name them all "Patrol Point" followed by a set of parentheses with the patrol point index within it, such as "(0)" for the first point and "(1)" for the next.

Our code is going to use that index value to properly order the patrol points in an array, so it's important that we always get it right.

Luckily for us, Unity will automatically do this sort of numbering for us when we copy-paste GameObjects.

Try it yourself. Create a Cube in the scene, and with Ctrl+C and Ctrl+V, copy and paste it a few times. You'll notice that Unity automatically renames the new cubes, adding a "(1)" after the first copy you create and then a "(2)" after the next and so on.

We'll use this to our advantage to make it easier to plop down a bunch of patrol points. We can just name the first patrol point "Patrol Point (0) ", and then whenever we copy and paste it, Unity will increase the index value for us. We'll still have to manually change the numbers if we ever want to add or remove a patrol point in the middle of the sequence, though. We'll just have to suck it up and deal with that.

Let's set up the GameObject for a Patroller so we can see this in action.

We'll use a system similar to that of the Player. A base, empty GameObject aligned with the floor (a Y position value of 0) will hold all the other GameObjects. We'll have a model within, which we'll be rotating instead of the base GameObject:

- Create an empty GameObject named "Patroller". Position it somewhere by your Player and make sure its Y position stays at 0.
- Create a Cube, name it Model Base, and make it a child of the Patroller. Set its local position to (0, 1.5, 0) and its scale to (3, 3, 3). The 1.5 Y position keeps the bottom touching the floor nicely.

- Since this will be a Hazard, check the Is Trigger field of the Model Base's Box Collider component.
- Create a second cube, name it Model Top, and make it a child of Model Base. Set its local position to (0, .5, .5) and its local scale to (.2, .2, .7). This will stick out at the top and front of the model, much like with the player, to give us better indication of where it's pointing. We'll also delete its Box Collider component so the player isn't killed by this little protruding piece (we'll show some mercy).
- Set the layer of the Patroller to Hazard, and when Unity pops up a box asking to change the layer of children, agree. We want the Patroller and all of its children to be in the Hazard layer.
- Give the root Patroller GameObject a Rigidbody component, and make it a kinematic one by checking the Is Kinematic box.
- Add a Hazard script component to the root Patroller GameObject. It will automatically use the box collider present in its child, the Model Base, to detect collisions that kill the player.
- You can create a new material for the model if you want. I'll use the same color for the patrollers as well as the wanderers we'll be making later, so I'm giving the material a more generic name: MobileHazard. I'll color it some kind of pink, with a hex value of EF7796. Of course, apply that material to the Model Base and Model Top cubes.

The model will look something like shown in Figure 18-2.

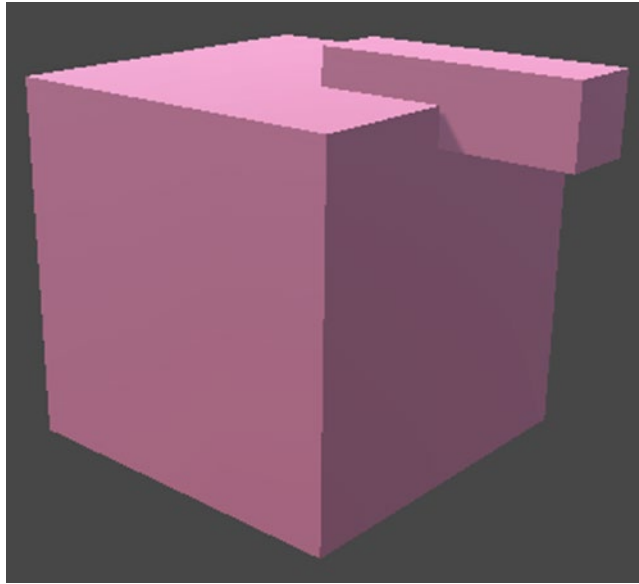


Figure 18-2. *Our Patroller model*

Now let's set up patrol points so we'll have them ready when we start testing the code that detects them. Just create an empty `GameObject` that's a child of the `Patroller` with `Alt+Shift+N` (`Option+Shift+N` for Mac users) while selecting the `Patroller`. As we discussed, name it "Patrol Point (0)". That'll be the first point. Leave it right where the `Patroller` `GameObject` is, with a local position of `(0, 0, 0)`. That way, the patroller will always double back on its initial Scene position.

Then, copy and paste the `Patrol Point` and position the new one at the next position you want the patroller to run to. The index in the name should go up by 1 point automatically. If you want, keep doing that until you've mapped out a path you're satisfied with (or just leave it at two points for a straight line). Remember, after reaching the last point (the patrol point with the highest index number), the patroller will go back to "Patrol Point (0)" and do it all again.

If you've created four patrol points, your Hierarchy should look something like Figure 18-3.

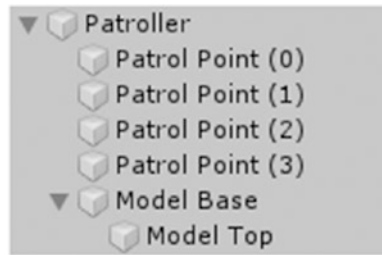


Figure 18-3. *The hierarchy of our Patroller GameObject after we've created four patrol points*

Detecting Patrol Points

Now we can start coding the Patroller. Create a script called Patroller and attach an instance of that new script to the root “Patroller” GameObject.

Before we implement anything else, let's do the part that relates to arrays: set up the patrol points. We're going to handle a couple of different firsts here, so bear with me.

We'll declare an array of Transforms called `patrolPoints`, nested directly in the Patroller script:

```
private Transform[] patrolPoints;
```

You'll notice it's private, which makes it not show in the Inspector, since we won't be setting it up that way.

Now we need to set the array up in the Start method. We're going to split this functionality up a little bit to keep things neat-looking. We'll do so by declaring a private method for us to use in our Start method. This private method will get all children of the Patroller and return only the ones with a name that starts with “Patrol Point”.

To do this, we'll be exploring some new concepts. First, we'll use the built-in method `GetComponentsInChildren<T>`, which you can call from a GameObject or a Component. The `<T>` part means that it takes a single generic type parameter, as we saw before with the `GetComponent<T>` method that we used in our Hazard script. It works much the same way: it's simply asking us what type of component we want to look for. It'll search all child GameObjects of the GameObject we call the method from and return an array storing all instances that it found of the given component type within the children.

Since every GameObject has a Transform that can't be removed from it, we can call `GetComponentsInChildren<Transform>` on our root Patroller GameObject to get a Transform for each child within it. Then, we can easily grab a reference to the

GameObject of each child, since Transforms have a “gameObject” member pointing to the GameObject that the Transform belongs to. Thus, finding all of the Transforms is just as well as finding all of the GameObjects.

Then we need to go over each item in that array and select only the ones with the correct name (the patrol points).

This is where we need a **List** and a **loop**.

The List is our first example of a class that takes a generic type. Technically, it should be List<T> since it takes one generic type parameter.

A list is an array that’s generally a bit less performant, but has less of the limitations that arrays have.

Arrays have to be created with a length. You must abide by that length from that point on. You’ll have “Length” items in the array, no more and no less. You can set items to null if you don’t need them, but they’ll still be there in the array.

A List doesn’t need to be created with a length in mind, and it allows you to add and remove items on the fly. This is perfect for us. We have no idea how many patrol points there will be at first, so we very well can’t create an array to store them in.

Aside from that difference, Lists operate like arrays. You still index them with a set of square braces [] to grab the items within. They’ll still throw an error if you try to grab an index that doesn’t exist.

We’re going to create a List, fill it with just the patrol point Transforms, and then return that List in this private method.

The loop is what we use to go through all of the Transforms in the initial array so we can add only the patrol points to the List.

We’ll get to that in a second, but let’s finally declare this method and get ready to add our loop. You know how generics work for methods, but this is the first time you’ll see them for class types:

```
//Returns a List containing the Transform of each child
// with a name that starts with "Patrol Point ("
private List<Transform> GetUnsortedPatrolPoints()
{
    //Get the Transform of each child in the Patroller:
    Transform[] children = gameObject.GetComponentsInChildren<Transform>();

    //Declare a local List storing Transforms:
    List<Transform> points = new List<Transform>();
}
```

Here, we declare a private method called `GetUnsortedPatrolPoints`. The return type is a `List`. When supplying a generic class as a type, you need to write out the `<T>` part as well. That's how the compiler knows what sort of object should be stored in the `List`. This way it knows that when we call this method, we'll get a `List` of `Transforms` returned to us.

Within the method, we use the `GetComponentsInChildren` method, as we talked about earlier. All we have to supply is the generic type parameter, which is `<Transform>`. We create a local variable called “children” to store this resulting array of `Transforms`.

Then, we declare a new local `List<Transform>` to store just the points in. This line looks somewhat redundant: `List<Transform>` comes first to signify the variable type, and then shortly after, we're typing it out again to construct a new instance.

There's a little “syntax sugar” you can use to avoid this, if it hurts your eyes so badly. The “`var`” keyword can replace the variable type, which makes the compiler simply figure out what the type is meant to be itself. In this situation, it's easy to do so: we're assigning a value to the variable immediately after, and that value is obviously going to be a `List<Transform>`, right?

```
var points = new List<Transform>();
```

When I say “syntax sugar,” I mean that this is purely in the syntax: it's not changing the functionality of our variable at all. It still stores a `List<Transform>` and only that. It's just that we're telling the compiler we're too lazy to type the name out ourselves and asking it to figure it out for us.

The “for” Loop

Now we're set up to do our **loop**. Simply put, a **loop** is a block of code that runs multiple times. There are different kinds of loops for different purposes, but the one we're using is a **for loop**.

Let's look at a simple example that's not exactly what we need, but it shows a basic use of the for loop:

```
for (int i = 0; i < 5; i++)
{
}
```

First, we give the “for” keyword and then a set of parentheses (). Within the parentheses, we have three separate tiny statements, all declared on the same line, but each one still separated by a semicolon “;” like they normally are.

The first statement is the **initializer**. It’s a variable declaration that occurs at the start of the loop. We declare a variable named “i”. This variable is initialized when the loop begins and only exists within the code block of the loop. You can’t access it outside of the loop.

The second statement is the **condition**. We’ve written “i < 5”. This is just a bool expression – something that returns true or false.

The third statement is the **iterator** which is just “i++”. This is a slightly shorter way of typing “i += 1”. In fact, if you really had your mind set on needless typing, you could type “i = i + 1”. But the shortest way is “i++”, so that’s what you’ll see most often in other people’s code.

So what is this loop doing?

First, the initializer code runs one time at the start of the loop.

Then, this recurring process happens any number of times:

- The condition is evaluated (i < 5).
- If the condition is true, the code inside the block is run, then the iterator is run (i++), and then this process repeats – back to the condition again.
- If the condition is false, break out of the loop.

In the example, this means the code in the block runs five times. After each iteration of the loop, the variable “i” increases by 1. Once “i” becomes 5, the condition is no longer true, so the loop stops iterating.

When the loop finishes, the code beneath it will continue running as normal.

At any point within the loop, we can access “i” to get its current value.

You might be able to guess how we can apply that to our current problem.

We want to loop over every item in the “children” array of Transforms, operating once on each item. We’ll use the “i” variable to refer to the current item in the loop – the index we want to grab from the array. It starts at 0 and goes up by 1 each iteration, which is perfect to go through each item in the array.

To ensure that we operate on each index in the array, we just need to know how many items are in the array, to make sure the loop stops as soon as we hit that last index (otherwise, we'll get an error for trying to access a nonexistent index). To do that, we use the "children.Length" value in the condition:

```
for (int i = 0; i < children.Length; i++)
```

This is the standard way to iterate over every item in an array or list and perform some action on each of those items individually. To get the item, we just use "i" as the index: "children[i]".

Now, what are we doing to each item in our loop?

What we need to do is check if the GameObject name starts with "Patrol Point (" and, if so, add it to the "points" List we made earlier.

This is how our method looks once we've added our for loop:

```
private List<Transform> GetUnsortedPatrolPoints()
{
    //Get the Transform of each child in the Patroller:
    Transform[] children = gameObject.GetComponentsInChildren<Transform>();

    //Declare a local List storing Transforms:
    var points = new List<Transform>();

    //Loop through the child Transforms:
    for (int i = 0; i < children.Length; i++)
    {
        //Check if the child's name starts with "Patrol Point (":
        if (children[i].gameObject.name.StartsWith("Patrol Point ("))
        {
            //If so, add it to the 'points' List:
            points.Add(children[i]);
        }
    }

    //Return the point List:
    return points;
}
```


Within the loop, we use “children[i]” to get the current child Transform in the loop. We reach into that and access the .gameObject and then reach further to access its name. This is a string, which has a handy instance method “StartsWith”. This simply returns true if the name starts with the given string parameter.

If so, we reference our “points” List and call its instance method “Add”. This takes one parameter, which is, of course, the item to add to the List. The item will be added to the end of the List (meaning it has the highest index).

Of course, we wouldn’t want to forget to actually return the list when we’re done, so we add that “return points;” statement to the bottom.

At last, we’ve made our method. Now we just have to put it to use.

Sorting Patrol Points

We have a method to get an unsorted List of patrol points. Now we need to get that list and, once again, employ a for loop. Iterating over each patrol point, we’ll isolate the index out of the patrol point name, convert it to an integer, and store the patrol point by that index in our “patrolPoints” array.

Remember, our main goal is to set up “patrolPoints”, the array we declared in the script to store the properly sorted array of patrol points.

So let’s declare that Start method and see what it looks like:

```
void Start()
{
    //Get an unsorted list of patrol points:
    List<Transform> points = GetUnsortedPatrolPoints();

    //Only continue if we found at least 1 patrol point:
    if (points.Count > 0)
    {
        //Prepare our array of patrol points:
        patrolPoints = new Transform[points.Count];

        for (int i = 0; i < points.Count; i++)
        {
            //Quick reference to the current point:
            Transform point = points[i];
```

```

        //Isolate just the patrol point number within the name:
        int closingParenthesisIndex = point.gameObject.name.
        IndexOf(')');

        string indexSubstring = point.gameObject.name.Substring(14,
        closingParenthesisIndex - 14);

        //Convert the number from a string to an integer:
        int index = Convert.ToInt32(indexSubstring);

        //Set a reference in our script patrolPoints array:
        patrolPoints[index] = point;

        //Unparent each patrol point so it doesn't move with us:
        point.SetParent(null);

        //Hide patrol points in the Hierarchy:
        point.gameObject.hideFlags = HideFlags.HideInHierarchy;
    }

    //Start patrolling at the first point in the array:
    SetCurrentPatrolPoint(0);
}
}

```

First, we use the “var” keyword again in our declaration of a local variable storing the result of calling that private method we just wrote. Just like that, we’ll have all of our Patrol Point GameObjects stored in a `List<Transform>`.

Pressing on, we see our first use of the `List<T>.Count` member. This is equivalent to the `.Length` member of an array, but for a `List`, it’s called `Count`, and resembles the number of items currently stored in the `List`. Using this member, we declare an “if” that ensures there’s at least one item in the list before we press on.

Now that we have “`points.Count`” to tell us how many patrol points there are in total, we can pass that as the length of the “`patrolPoints`” array.

This initializes the array with just the right amount of items: one for each patrol point in the list. They’re all null at first, but we’re about to set each one in the for loop we declare next.

The loop iterates over each point in “points”, using “points.Count” as the total number of iterations to perform.

This time, we do a common operation at the start of the loop code block:

```
//Quick reference to the current point:
Transform point = points[i];
```

This creates a local variable storing the current point we’re iterating on. Now, instead of typing “points[i]” to refer to the current point, we just type “point”.

Grabbing an item from a collection takes a little longer than referencing an existing variable, so if you’re going to access an item multiple times in your loop, it’ll be faster to store it in a local variable instead. Not only that, but it’s just a little easier to type.

The next section involves some new concepts regarding the manipulation of strings:

```
//Isolate just the patrol point number within the name:
int closingParenthesisIndex = point.gameObject.name.IndexOf('');

string indexSubstring = point.gameObject.name.Substring(14,closing
ParenthesisIndex - 14);
```

The first line declares an integer value, using the string .IndexOf method.

To understand what this does, let’s learn a little more about strings.

Strings are just collections of **characters**, where each character is a single letter, number, or symbol in the string. Technically, these characters are resembled as the “**char**” data type.

Strings actually function something like an array or a list, in that you can index them to grab a specific character from them. And just like an array, their index starts at 0 for the first character and then goes up by 1 per character. They even have a .Length member you can use to get the total number of characters in the string.

The .IndexOf method takes one parameter: a char or a string. It searches the string from left to right until it finds that parameter occurring within the string. If it finds it, it returns the index of the char – or if a string was given, it returns the index that the first char in that string occurs at.

The parameter we give is ‘)’. You’ll notice that’s using single quotes, not double quotes as you would normally use when declaring a string. This is how you declare a char. Just use single quotes and, of course, only put one character within them.

So that variable declaration is essentially asking “What index is the ‘)’ character placed at?”

Once we have that, we can use another handy string method: Substring.

This method takes a start index and a count, both integers. It returns a piece of the string (a substring, so to speak). It starts at “startIndex” and returns a new string containing “count” characters from that point of the string and onward.

To put it simply, it’ll return a piece of the string, starting at “startIndex” and ending at “startIndex + count”.

Our goal is simply to get a new string containing only the characters between the ‘(’ and the ‘)’ in the Patrol Point name. If you count out the characters in “Patrol Point (”, starting at 0, you’ll get 13 (remember, the spaces count too). That puts you right at the index of the ‘(’ character. So just go one character higher, and we’ll start at the character immediately after.

The count is a little trickier. We could have patrol points with double-digit or, God forbid, even triple-digit indexes. So we can’t just say 1 or 2 or 3. We need a string with just the numbers of the index in it – it can’t include the ‘)’ at the end or anything else. We need to be precise.

This is why we needed to grab the index of the closing ‘)’ character just before. Using that, we can do a little math: the index of the ‘)’ character minus 14 is the number of characters we want to get. For example, if index 13 is the ‘(’ and index 15 is the ‘)’, then there’s only one character between them – index 14. So 15 – 14 gives us a count of 1. We grab one character, starting at index 14, which means we’re just grabbing the character at that index.

Okay, moving on, we see why we needed this data in the first place:

```
//Convert the number from a string to an integer:
int index = Convert.ToInt32(indexSubstring);

//Set a reference in our script patrolPoints array:
patrolPoints[index] = point;
```

We declare an integer for the index – not a string anymore, because as you’ll see, we plan on passing this index to the array to assign the item value. We can’t do that with a string, even if it only contains numbers.

To convert the string of numbers to an integer, we use `System.Convert.ToInt32`. This is a native C# method that takes a string and gives back an integer. Don’t worry about why it’s called “ToInt32” instead of just “ToInt” – that’s just technical stuff.

If the string has any non-number characters in it, an error will be thrown. That’s why we had to be all picky about how we created the substring.

Note that **if you don't have a “using System;” declared at the top of your script file, this won't work!** Convert comes from the “System” namespace. You'll have to either declare that using or, if you'd rather, just change the “Convert” to “System.Convert” instead. Otherwise, you'll get a compiler error.

Next, we have this bit:

```
//Unparent each patrol point so it doesn't move with us:
point.SetParent(null);

//Hide patrol points in the Hierarchy:
point.gameObject.hideFlags = HideFlags.HideInHierarchy;
```

The first line is somewhat self-explanatory. We call the SetParent method of the point (which is a Transform). This takes one parameter, the Transform we want to set the parent to. We pass in null, which means “no parent at all.” This makes sure the patrol points don't move with the Patroller when it moves.

The next line is another new concept.

Unity uses this enum “HideFlags” to let us specify certain little things about the destruction and visibility of objects. It can hide scripts from the Inspector so they don't show at all. It can also hide GameObjects from the Hierarchy, which is what we're asking it to do here: we set the hideFlags member of each point GameObject to “HideInHierarchy”, which does just as the name suggests – makes the GameObject no longer show up in the Hierarchy.

This is just to clean things up for us while we're playing. Since the patrol points are no longer parented to their respective Patrollers in-game, they'll all be spread about willy-nilly in the Hierarchy when we're playing. This can make it somewhat cluttered (particularly if we had lots of different patrollers in the same scene), so we fix that by hiding them.

Of course, Unity will revert everything back to the way it was once we stop playing, so all of our patrol points will be visible in the Hierarchy again.

The final line is the calling of a method that we're going to declare in a bit:

```
//Start patrolling at the first point in the array:
SetCurrentPatrolPoint(0);
```

It does just what it says. It sets our current patrol point – the one the Patroller will be moving toward – to the first one in the patrolPoints array. Like I said, we'll be declaring that in a bit.

And that's that! It took the learning of a fair number of new concepts, but our patrol points should be setting themselves up properly in-game. We won't be able to see any effect until we make the Patroller move along the points, though, so let's get on that.

Moving the Patroller

Let's move back up to the top of the script to declare the variables we'll be needing to handle moving the Patroller. So far, the only variable we have is the "patrolPoints" variable. Let's give it some company by adding in these variables – the new ones are in bold, so make sure not to declare a second "patrolPoints" variable:

```
//Consts:
private const float rotationSlerpAmount = .68f;

[Header("References")]
public Transform trans;
public Transform modelHolder;

[Header("Stats")]
public float movespeed = 30;

//Private variables:
private int currentPointIndex;
private Transform currentPoint;

private Transform[] patrolPoints;
```

The very first variable declaration is another new concept.

It's called a "**const**". It's short for **constant**. Shortly put, it means "a variable that can't be changed."

It's declared just like a normal variable, but of course, you throw that "const" keyword before the type name. This marks the variable as constant. You have to assign a value to it right there when you declare it, and if you ever try to change its value, a compiler error will prevent you.

This is used to ensure that a variable that shouldn't be changed won't be changed. This way, you can assign a name to some value you're planning on using within your code, and if you ever need to change it, you can do so in one place. However, it makes it clear that the value is not meant to change within the code.

We'll get into the purpose of this const variable later, when we start rotating the Patroller to face the direction of movement. For now, let's move on.

You know about the references, since we've done them a few times. The first one points to our own Transform – slightly faster than using “transform” – and the second one will point to the model Transform. That's so we can rotate just the model, not the root GameObject, like we do with the player.

We then have the movespeed, which is self-explanatory, and then some private variables:

- **currentPointIndex** is an int resembling the index in the “patrolPoints” array that we'll use to get the current patrol point.
- **currentPoint** is a reference to the Transform of the patrol point we're currently moving toward.

Let's move on to declare our “void Update()” method, where the real magic happens. Declare a “void Update()” if there isn't one already and put this code in it:

```
//Only operate if we have a currentPoint:
if (currentPoint != null)
{
    //Move root GameObject towards the current point:
    trans.position = Vector3.MoveTowards(trans.position,currentPoint.
    position,movespeed * Time.deltaTime);

    //If we're on top of the point already, change the current point:
    if (trans.position == currentPoint.position)
    {
        //If we're at the last patrol point...:
        if (currentPointIndex >= patrolPoints.Length - 1)
        {
            //...we'll set to the first patrol point (double back):
            SetCurrentPatrolPoint(0);
        }
        else //Else if we're not at the last patrol point
            SetCurrentPatrolPoint(currentPointIndex + 1);
        //Go to the index after the current.
    }
}
```

```

//Else if we're not on the point yet, rotate the model towards it:
else
{
    Quaternion lookRotation = Quaternion.LookRotation((currentPoint.
    position - trans.position).normalized);

    modelHolder.rotation = Quaternion.Slerp(modelTrans.rotation,
    lookRotation, rotationSlerpAmount);
}
}

```

First, we move the root Transform toward the point, using the method `Vector3.MoveTowards`. This takes three arguments: a `Vector3` for the position being moved, a `Vector3` for the position to move toward, and a float for the distance to move. It moves the first vector toward the second by that float amount and returns the result. If the movement amount is enough to overshoot the target point, it'll just return the target point instead – it won't go past it. That way, we won't have any awkward situations where the patroller shoots on past its patrol point.

After performing the movement, we do an “if” asking if our position is equal to the point position – in other words, if we've already reached the point. We know the `MoveTowards` method will return exactly the `currentPoint.position` once we reach it and won't overshoot that point, so we can safely compare the positions this way.

If we have reached the point, we then check whether our current point index is equal to or greater than the length of the `patrolPoints` array – minus 1, to account for the indexes starting at 0. In other words, we're asking “Are we at the last point?” If so, we call this `SetCurrentPatrolPoint` method (which we still haven't declared) to set the patrol point to the first patrol point by giving it the index 0.

Otherwise, if we're not at the last point, we call the same method, but pass the `currentPointIndex + 1` to use the next patrol point in the array.

If we have not yet reached the point, we'll just rotate our model toward it. We do this with much the same method as we used to, Slerping the current model rotation toward the rotation required to look at the point. We use that const variable we declared earlier, `rotationSlerpAmount`.

The main difference here is the method we use to get our `LookRotation`. This is vector stuff, and we aren't going to get too deeply vested in it during this first example project. We'll have more room to play with directions and whatnot in later chapters.

For now, just know that when trying to get a direction to point from a Vector “**from**” to a Vector “**to**”, you simply do this: “(to - from).normalized”. In our case, “to” is the current point, and “from” is our position: we want the direction **from** our position **to** the position of the point, so we end up with this:

```
(currentPoint.position - trans.position).normalized
```

As we’ve demonstrated before, Quaternion.LookRotation converts this to a Quaternion pointing along that given direction.

Anyway, let’s move along. We have one final method we need to declare, the SetCurrentPatrolPoint method.

This method is just a simple way to make sure we Don’t Repeat Ourselves. Whenever we change the current point, we’ll be doing so by index, but we need to also make sure currentPoint gets set. So we have this method to simplify things – it just automates the setting of currentPoint for us.

Declare it **anywhere in the script class**:

```
private void SetCurrentPatrolPoint(int index)
{
    currentPointIndex = index;
    currentPoint = patrolPoints[index];
}
```

Now if you play, everything should work out. The points find themselves on Start, the Patroller begins moving, and it properly rotates the model to face the current point while moving toward it.

Summary

This chapter went over a lot of new and important concepts and taught us some pretty handy tricks. We implemented a new type of obstacle and learned how to work with arrays and the “for” loop. We also worked with generic methods and classes (resembled by the angle brackets “<>”) a little more. Some things to remember are as follows:

- An **array** stores many instances of a specific data type in a single value. We refer to one of these instances as an “item” in the array.

- To access a specific item within an array, square braces [] are used. This is called an **indexer**. Within the square braces, we provide an integer for the **index** of the item we want to access. The index is a number value used to represent an item in the array.
- The “**Length**” member of an array returns the number of items stored in the array. This cannot be changed after the array is created.
- The first index in an array is always 0. The last index is the Length of the array minus 1.
- Serialized arrays will show in the Inspector, allowing you to set their length and assign a value to each member.
- A **List** is an array that does not have a fixed Length. With a List, you may add and remove items on the fly, and the “Length” member is replaced with “Count”, which returns not the maximum number of elements that can be stored, but rather the number of items currently stored.
- The “gameObject.hideFlags” member can be set to hide a GameObject from the Hierarchy.
- In general, a **loop** is a block of code that can run multiple times based on the situation.
- A **for loop** is a block of code that commonly **declares an iterator variable** to store an integer value and then repeatedly runs the code in its code block and, each time, performs some operation on the iterator – usually either increasing it by 1 or decreasing it by 1. You’ll see and use the for loop all over the place and for various purposes, but its most common use is to iterate over each item in an array and perform some code on it, using the iterator as the item index.
- The `GameObject.GetComponentInChildren<T>` method takes a single generic type parameter (the “T”) and can be called to return an array of type T. This array stores references to all instances of components of type T found in children of the GameObject that the method was called from.

- To get all GameObjects that are a child of a given GameObject, you can just get the Transforms of all children with a call to `GetComponentInChildren<Transform>()` and then use the `Transform.gameObject` member to grab a reference to the GameObject that each returned Transform belongs to.
- The `IndexOf` method can be called when reaching into a string to return the index that a given string occurs at within that string. For example, `"hello".IndexOf("e")` would return 1. Character `"h"` is at index 0, and `"e"` is at index 1.
- The `Substring` method can be called when reaching into a string to return only a portion of the text contained in the string. It takes two parameters: the index of the character to start at and the number of characters afterward to include in the returned string.
- A variable that is declared with the **`const`** keyword is a variable that cannot be changed from whatever value it is initially assigned.

CHAPTER 19

Wandering Hazards

In this chapter, we'll implement an obstacle type that is placed within a wander region, which is pretty much an invisible box, and will randomly move to a new point within the region at random times. They'll look like the Patrollers we implemented in the last chapter, but their movement will be constrained within the wander region they're associated with. To avoid sudden movement that kills the player in frustrating ways, we'll make the wanderers turn to point in the direction they're going to travel next, wait a little, and then move. This gives the player a little time to react.

Wander Regions

The Wanderer script will be used to create obstacles that are confined to a box, randomly selecting a new point to run to within that box. When they select a new point, we call it **retargeting**. Every time they retarget, they turn to face that new point, wait a certain duration, and then begin moving toward it. Once they reach it, they just stand there and wait for their next retargeting.

Of course, they'll use the Hazard script to kill the player on touch.

First of all, we need to define the boxes they're confined to.

We'll create a script to resemble one of these boxes, called a wander region. Go ahead and make a new script named `WanderRegion`, and while we're at it, we'll make a `Wanderer`. The `Wanderer` is the script that goes on the obstacle itself, the thing that moves and acts as a hazard.

Open up the `WanderRegion` first, and let's get started on it.

Defining the box of a region is simply a `Vector3` "size" variable. The box is positioned on the `transform.position` of the `WanderRegion` and sized equal to this variable.

The region does two things:

- Defines a space that Wanderers will attach themselves to, providing them with a method to get a random point within that space.
- Contains all Wanderer obstacles as its children. On Start, it finds all of these children and gives them a reference to itself (the region). The Wanderers will use this reference to get a new point whenever they need to retarget.

Let's get to the code – place this **within the WanderRegion script class**:

```
[Tooltip("Size of the box.")]
public Vector3 size;

public Vector3 GetRandomPointWithin()
{
    float x = transform.position.x + Random.Range(size.x * -.5f, size.x * .5f);
    float z = transform.position.z + Random.Range(size.z * -.5f, size.z * .5f);

    return new Vector3(x, transform.position.y, z);
}

void Awake()
{
    //Get all of our Wanderer children:
    var wanderers = gameObject.GetComponentsInChildren<Wanderer>();

    //Loop through the children:
    for (int i = 0; i < wanderers.Length; i++)
    {
        //Set their .region reference to 'this' script instance:
        wanderers[i].region = this;
    }
}
```

We've just learned about arrays and the `GetComponentsInChildren` method, so the `Start` method we've written is pretty familiar. As you'll recall, the "this" keyword can be used to refer to the object instance that the code is running from – in this case, the

WanderRegion instance. So all we're really doing is finding all of the Wanderers that are children of the Region, looping through them, and giving them a reference to this WanderRegion.

We're doing this in void Awake() because it occurs before Start. This way, we can expect that our Wanderers will already have their .region set when their own Start method is called.

Since we haven't written that variable in the Wanderer script yet, we'll be plagued by the error in the compiler until we do.

We'll write the rest of the Wanderer logic later, but we need to fix that error before then. Just open up the Wanderer script and declare this variable within the script class:

```
[HideInInspector] public WanderRegion region;
```

We declare it as public, since the WanderRegion script needs to be able to access it. However, we don't expect to be setting this member manually in the Inspector – the whole point of detecting them in the WanderRegion script is to avoid this. So we ought to give it the [HideInInspector] attribute so it doesn't show up.

Moving on, the GetRandomPointWithin method will be called by the Wanderers (through their .region reference). It uses a little random number generation to grab a point somewhere within the box.

The method we use to generate a random number is called Random.Range.

It's pretty simple: give it a minimum value and a maximum value. It gives back some random value between the two.

Since the box of the region is meant to be centered on its Transform, the X and Z position of the Transform is always used as a base. Then, our range is defined as “anywhere between half the size in the negative direction and half the size in the positive direction.”

We only care about the X and Z axes because the Wanderers should remain at the same Y position throughout the game (just like everything else, pretty much).

When we return the point, we use the X and Z values we randomly generated, but pass the Y position of the box Transform. Thus, we need to make sure our regions are always touching the floor, or they'll give incorrect Y positions to their Wanderers.

This is all we need for the WanderRegion script, but it kind of sucks. We can't see the box. That makes it hard to design levels. We'd have to measure the size manually, for example, with a cube scaled to the same size of the box, positioned at the same location.

Let's make a better solution instead.

A Basic Editor Extension

Unity gives its developers various ways to extend the editor itself. We can write code that changes how our Inspectors behave, draws gizmos to the Scene view, or even defines entirely custom windows (like the Inspector and Hierarchy, except developed by us instead of coming default in the engine).

We're going to explore a simple concept – a custom inspector that we can use to draw a box to the Scene view whenever a `WanderRegion` is selected.

Editor extension is a wide topic with many pitfalls that can catch you off guard. We're going to keep this lightweight and easy to implement – you've got enough on your plate anyway, right?

In spite of that, extensions to the Unity editor can be particularly useful in easing the flow of development. When working with a team, they can aid designers in dealing with tools created by the programmers.

All we want to do is make it so when our `WanderRegion` is being viewed in the Inspector, a box shows up in the Scene view, using the “size” member of the region.

Editor Scripts

When creating anything that extends the editor, you must do two things:

- Place the code that relates to editor extending in a script somewhere within a folder named “Editor” in your Assets. It can be buried down a hundred different folders, if you want – but at least one of them must be named “Editor”.
- In any script that extends the editor, write a “using UnityEditor;” statement at the top. This namespace contains the relevant classes for extending the editor.

In your Scripts folder, create another folder, named Editor.

In that Editor folder, create a script called `WanderRegionInspector`. The name doesn't have to be exactly that, but when writing a custom inspector, I find it's a good practice to simply name it the same as the script you're writing the inspector for and then add “Inspector” at the end.

Open it up and add that “using UnityEditor;” namespace at the top to ensure that we have access to the names we're going to be using.

Custom Inspectors

Now we need to modify the script class in two ways to make it no longer a script component, but instead a custom Inspector class.

The first is a `CustomEditor` attribute, with a single parameter pointing at the “typeof” the `WanderRegion`. Remember, the “typeof(...)” syntax is necessary whenever referring directly to a type (that’s not a generic type parameter).

The second modification is the class we inherit from. Instead of `MonoBehaviour`, we’ll change it to `Editor`.

In the end, your class will be declared like so:

```
[CustomEditor(typeof(WanderRegion))]
public class WanderRegionInspector : Editor
{
    //...
}
```

Now we can write the contents.

We can declare variables, but they’ll get reset whenever the Inspector switches away from our script, so we can’t really store anything persistent.

In our case, we don’t need variables. We have our “size” variable in the `WanderRegion`, so we’ll use that. We just need to declare a built-in event method called `OnSceneGUI`.

This method runs whenever the scene GUI is drawing, while the Inspector is showing an instance of `WanderRegion`. Within the method, we can call other built-in Unity methods that let us draw stuff to the scene.

Accessing the Inspector Target

Before we begin, we need access to the `WanderRegion` that we’re inspecting. The `Editor` class that we’ve inherited from provides us with a variable pointing to just that. It’s called “target”.

But the “target” variable has a funny little quirk: its type is “Object”. In other words, the compiler views it as the most basic form of Unity object – any object, of any more specific type. So whenever we access it, the compiler won’t know that we’re expecting it to be a `WanderRegion`.

The fix is simple: we just explicitly cast it.

Instead of typing “target”, we type “(WanderRegion)target”.

That’s somewhat tedious if you’re accessing the target many times in your inspector code. A workaround is to declare a simple property that performs the typecast for us. Write this declaration **within the WanderRegionInspector class**:

```
//Quick reference to target with a typecast:
private WanderRegion Target
{
    get
    {
        return (WanderRegion)target;
    }
}
```

Instead of referencing “target” with a lowercase T, we reference “Target” with an uppercase T. This gets us the property instead of the inherited variable. Sneaky, isn’t it?

Since the property is of the WanderRegion type, there’s no need to cast it when we reference it. The casting is already done in the getter.

Drawing to the Scene

Anyway, now for the functionality of our custom inspector. We won’t actually change the way the script is shown in the inspector itself (although we could). We’ll just draw to the scene while the script is showing in the Inspector.

Add this code somewhere **beneath the Target property** declaration:

```
//The height of the box display.
private const float BoxHeight = 10f;

void OnSceneGUI()
{
    //Make the handles white:
    Handles.color = Color.white;

    //Draw a wireframe cube resembling the wander region:
    Handles.DrawWireCube(Target.transform.position + (Vector3.up *
        BoxHeight * .5f),new Vector3(Target.size.x,BoxHeight,Target.size.z));
}
```

First, we declare a const variable depicting the height of the box display. We'll do 10 units. The height is just visual – remember, the random points we get from the region are always going to have a Y position equal to that of the WanderRegion itself.

Then, we declare the built-in method OnSceneGUI.

This is another method like Update and Start – Unity will call it for us when the scene GUI is updating. Within this method, we can call certain other built-in methods to draw to the scene.

The Handles object provides us with various means of drawing (you guessed it) handles to the scene. Handles can be visual elements or things that can be interacted with by clicking them.

First, we set the Handles.color variable, which we can set at any time to change the color of our handles before we draw them. We set it to Color.white, a shorthand property for declaring a new Color resembling white. Many of these shorthand properties exist for other basic colors, like red, green, or blue. It's just for convenience and readability.

The method we call next is telling Unity to draw a wireframe cube. Wireframe means it's not a solid cube – it's just the edges, a visualization of a box.

We do a little vector trickery to get the position of the cube. Since the position is the center of the cube, we want to draw it such that the bottom is touching the floor – the Y position of the WanderRegion.

So we start at the position of the WanderRegion, which we get with Target.transform.position. Then, we “go up” by half of the BoxHeight const we declared earlier.

The Vector3.up is another one of those shorthand properties. You'll see these often if you read code written by other Unity programmers. It just returns “new Vector3(0, 1, 0)”. Multiply that by a float – like half of the box height – and we get a vector with that float in the Y axis and 0 in the X and Z axes.

This is how people often do vector math, so that's why I've written it this way. Now that you know this, you'll have an easier time reading other people's code.

You could just as easily change it to this:

```
Target.transform.position + new Vector3(0, BoxHeight * .5f, 0)
```

They both do the same thing in a different way.

Moving on, the next parameter is the size of the box. We construct a new Vector3 here (no shorthand properties can save us now) and give it the size.x and size.z from our Target, which will be set in the Inspector. Of course, the height is BoxHeight – we don't care about the Y axis of the WanderRegion.size.

Save that and navigate to Unity. Now, create an empty GameObject, name it Wander Region, and attach a WanderRegion script instance to it.

At first, the size will be (0, 0, 0), so change the X and Z axes to something higher than 0 and you should see the box show up, centered around your Wander Region GameObject. Figure 19-1 shows how it will look in an empty scene.

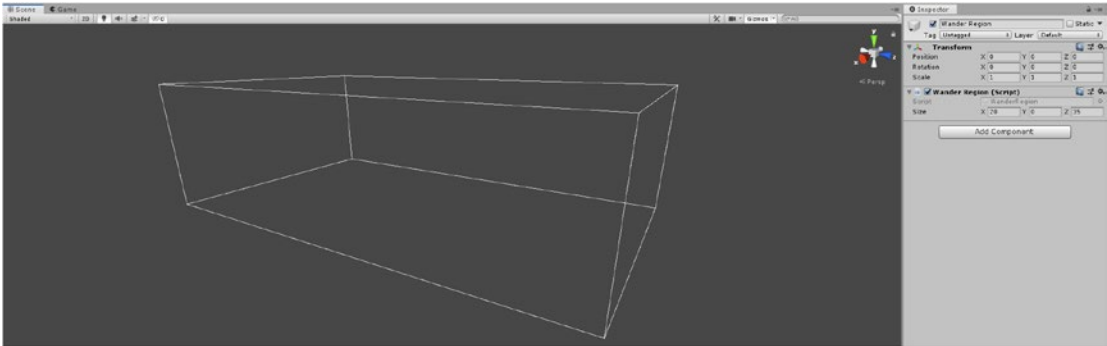


Figure 19-1. A Wander Region box showing in the Scene view. The “size” property is set to (20, 0, 35), making the box 20 units wide and 35 units long

The box will dynamically update to match the size in the Inspector, and it’ll stay 10 units high no matter what the size.y value is set to, as intended. Now we don’t have to do any tedious tricks to visualize how large our regions will be when we’re sizing and placing them.

Wanderer Setup

Let’s set up a Wanderer obstacle so it’s ready to test – then we’ll write the script for it.

The Wanderer will have a very similar hierarchy to the Patroller. We’ll use the same model too:

- Create an empty GameObject (Ctrl+Shift+N). Name it Wanderer.
- Attach a Hazard script, a kinematic Rigidbody, and a Wanderer script.
- Copy-paste the Model Base and its child, Model Top, from the Patroller we made earlier. Make it a child of the Wanderer, and set its local position to (0, 1.5, 0). Leave the scale as is.

- The Model Base should have Is Trigger checked on its Box Collider component already, but if not, make sure to do that.
- Make sure the Wanderer and all of its children are in the Hazard layer.

The Wanderer needs to be a child of the Wander Region it belongs to as well, so don't forget to drop it in there.

Wanderer Script

We've put it off long enough – it's time to tackle the script resembling a wandering obstacle.

Let's detail how they'll behave as far as the programming is concerned.

At any given moment, a Wanderer will either be Idle, Rotating, or Moving:

- While **Idle**, they just stand there and wait.
- While **Rotating**, they're not moving, but are turning toward the next position they've targeted. After they finish turning, they wait a certain extra amount of time to give the player time to react to the final rotation. Then, they begin Moving.
- While **Moving**, they travel toward the point (which they will now be directly facing) until they reach it, where they stop and become Idle again.

We'll declare a State enum **directly inside the Wanderer script class** and a private member of the State type. As shown in the following, we'll put them next to the "region" member we declared earlier – the new code you should add now is shown in bold:

```
private enum State
{
    Idle,
    Rotating,
    Moving
}

private State state = State.Idle;

[HideInInspector] public WanderRegion region;
```

This is a common practice to handle objects that need to toggle in and out of different “modes” that designate different behavior. In our Update method, we’ll use the current value of “state” to determine what the Wanderer should be doing at any given moment.

Because State is a private enum declared inside the script class, we can only access it from this script class. Thus, there’s no need to name it “WandererState” or anything of the sort.

Beneath those variables, we’ll declare the rest of our variables:

```
[Header("References")]
public Transform trans;
public Transform modelTrans;

[Header("Stats")]
public float movespeed = 18;

[Tooltip("Minimum wait time before retargeting again.")]
public float minRetargetInterval = 4.4f;

[Tooltip("Maximum wait time before retargeting again.")]
public float maxRetargetInterval = 6.2f;

[Tooltip("Time in seconds taken to rotate after targeting, before moving begins.")]
public float rotationTime = .6f;

[Tooltip("Time in seconds after rotation finishes before movement starts.")]
public float postRotationWaitTime = .3f;

private Vector3 currentTarget; //Position we're currently targeting
private Quaternion initialRotation; //Our rotation when we first retargeted
private Quaternion targetRotation; //The rotation we're aiming to reach
private float rotationStartTime; //Time.time at which we started rotating
```

At this point, “trans”, “modelTrans”, and “movespeed” ought to be pretty self-explanatory.

The retarget interval is defined as a float for the minimum and another for the maximum. We'll use `Random.Range` to get a time between the two and `Invoke` a retargeting to happen in that amount of time.

The rotation time will be used to measure just the period of rotating toward the target point.

The `postRotationWaitTime` is the time it takes to start moving after the rotation finishes. So once a retargeting begins, it takes the sum of both these values before the obstacle actually moves.

The private variables are pretty much explained in their comments, and we'll go into further detail once we get to using them. Each one of these variables is set again whenever a retargeting occurs; they're all storing data that we'll need to properly perform a retargeting.

Handling the State

The rest is going to be a bit of fun. We're going to use method invoking and the changing of our states to create a sort of recurring loop between idle and rotating and moving.

First, let's declare some methods **in the Wanderer script, down beneath our variables**:

```
//Called on Start and invokes itself again after each call.
//Each invoke will wait a random time within the retarget interval.
void Retarget()
{
    //Set our current target to a new random point in the region:
    currentTarget = region.GetRandomPointWithin();

    //Mark our initial rotation:
    initialRotation = modelTrans.rotation;

    //Mark the rotation required to look at the target:
    targetRotation = Quaternion.LookRotation((currentTarget - trans.
    position).normalized);

    //Start rotating:
    state = State.Rotating;
    rotationStartTime = Time.time;
```

```

    //Begin moving again 'postRotationWaitTime' seconds after rotation ends:
    Invoke("BeginMoving",rotationTime + postRotationWaitTime);
}

//Called by Retarget to initiate movement.
void BeginMoving()
{
    //Make double sure that we're facing the targetRotation:
    modelTrans.rotation = targetRotation;

    //Set state to Moving:
    state = State.Moving;
}

```

Here, we declare a Retarget method to handle the assigning of a new target. Inside, we call that method we declared in the WanderRegion script to get a new random point, assigning it to our private variable “currentTarget”. We save our current rotation and, again, use LookRotation with the same “(to – from).normalized” formula to get the direction from the current position to the target position. That will be our target rotation, which we’ll reach over the course of the rotationTime.

To initiate rotation, we mark the time that it began, and we set our state to Rotating.

After, we invoke the method we declared just below this one, called BeginMoving. It will happen once the rotationTime and the postRotationWaitTime have both elapsed.

The BeginMoving method is just a few lines of code: set the state to Moving and ensure we’re pointing exactly at the target rotation, just to be safe in choppy situations where there’s a lot of time between frames.

With that in place, we just need to make sure that we initiate the process **in our Start() method**. We just need to call Retarget():

```

void Start()
{
    //On start, call Retarget() immediately.
    Retarget();
}

```

Reacting to the State

Now we just need the frame-by-frame logic that moves or rotates the Wanderer based on the current value of “state”. Put this **in your Update method**:

```
if (state == State.Moving)
{
    //Measure the distance we're moving this frame:
    float delta = movespeed * Time.deltaTime;

    //Move towards the target by the delta:
    trans.position = Vector3.MoveTowards(trans.
    position,currentTarget,delta);

    //Become idle and invoke the next Retarget once we hit the point:
    if (trans.position == currentTarget)
    {
        state = State.Idle;
        Invoke("Retarget",Random.Range(minRetargetInterval,maxRetarget
        Interval));
    }
}
else if (state == State.Rotating)
{
    //Measure the time we've spent rotating so far, in seconds:
    float timeSpentRotating = Time.time - rotationStartTime;

    //Rotate from initialRotation towards targetRotation:
    modelTrans.rotation = Quaternion.Slerp(initialRotation,targetRotation,
    timeSpentRotating / rotationTime);
}
```

With you being an expert programmer now, the movement is old news. We did the same thing with patrollers in the previous chapter: measure the delta (movement on this frame), and then use `Vector3.MoveTowards` to move the root Transform toward the target point by “delta” distance.

After, we make sure to stop moving once we reach the point by becoming Idle – again, MoveTowards won't overshoot the point, but we might as well save ourselves some unnecessary calculation and stop calling it when we know we're already there. Also, we use this moment to invoke the Retarget method again. We use Random.Range to get a random time within the retarget interval.

The rotation of the model is where things get a bit different – notably, we're using Slerp in a different way than we have before.

As you'll recall, Slerp takes one rotation and moves it toward another by a fraction of the difference between the two and returns the result.

Some examples are as follows:

- If that fraction is 0, there's no change, so we simply get the first rotation returned back as is.
- If the fraction is .5, a rotation halfway between the two is returned.
- Make it 1, and the rotation is moved all the way to the target, so we simply get the target rotation returned back as is.

Up until now, we've only used this by passing in the current rotation of some Transform as the first rotation and Slerping it toward a target rotation.

But that's not the only way to do it. We're making a game here, and we want to implement this mechanic in a way that makes it more fun. We're making the wanderers rotate toward their target to ensure that the player sees them do it so they can react to the rotation and get out of the way. So we want it to take a certain amount of time for the rotation to finish, consistently. The method we've used up until now doesn't really do that.

To counter this, we'll use the same two rotations every time we call the Slerp method and keep applying the result to the Transform. Before, we Slerped the current rotation toward the target rotation. Now, we've marked the initial rotation of the wanderers when they began the turn, and we're going to Slerp that initial rotation toward the target rotation.

The tricky part is figuring out what fraction we pass in as the third parameter. We've already set up the variables required to do this earlier. We know how long we want it to take to make the turn (rotationTime) and the time at which the turn began (rotationStartTime).

So we want the fraction we give to the Slerp to start at 0 when we first begin turning and then raise to 1 over the turn duration. In other words, start at the initial rotation, and change to the target rotation over the duration.

We do this by grabbing the “time spent rotating so far.” This is just “current time – begin time.” For example, if we began at 16.2 and the current time is 16.6, that’s $16.6 - 16.2 = .4$ seconds spent rotating so far.

Then, we just need to turn that into a fraction (a value between 0 and 1) of the total time we want it to take.

Whenever you have to ask “What fraction of the value Y is the value X?” you just do “X/Y”. In a more understandable example, let’s say we’re coding an RPG and we want to know what percentage of their maximum health our player character is at.

To get this, you’d just do “currentHealth/maxHealth”. This is a fraction, so a value between 0 and 1, but you can easily turn it to a percentage by multiplying it by 100 and then tacking a % sign on the end.

We’re doing much the same thing here. We’re dividing the “rotation time so far” by the “total rotation time desired,” thus converting it to a value that starts at 0 and raises to 1 over the rotationTime.

And that covers it all – we should have operational Wanderers now.

We never actually attached the Wanderer script to our GameObject. Now that it’s ready, go ahead and do that – remember, put it on the root Wanderer GameObject, not the model. Set the “trans” reference to the root Transform and set the “modelTrans” reference to that of the Model Base.

Ensure that the Wanderer is a child of a Wander Region with a big enough box to play around in, hit Play, and watch it go.

Summary

This chapter put another new obstacle into motion and taught us how to extend the Unity editor to implement a custom Inspector for a script.

- The Random.Range method takes two int or float parameters and returns a random value between the two.
- When you need to determine the time that has passed since a certain event occurred, you can set a float variable to “Time.time” when the event first occurs. Afterward, use “Time.time – floatVariable” to get the time, in seconds, since the event occurred.
- To calculate what fraction the float “X” is of the float “Y”, just divide “X/Y”. This will return .5 if X is half of Y or .75 if it is 75% of Y and so on.

- Scripts which extend the Editor must be placed somewhere within a folder we've named "Editor" in our Project window. You can have more than one Editor folder in your project if you like, and they can be nested in other folders.
- Scripts which extend the Editor rely on classes in the UnityEditor namespace, so you'll want to place a "using UnityEditor;" line at the start of your editor scripts.

CHAPTER 20

Dashing

To give the player a little more interesting movement, we're going to implement an option for them to quickly "dash" in the direction they're moving – or, more specifically, in the direction they have held down with the movement keys. If they're holding W and D, they'll dash diagonally forward and right, for example.

When dashing, the player will stop moving by normal means, giving control to the dash movement instead. Once they finish dashing, we'll revert control back to the normal movement, and we'll set the movement velocity to full power in the direction they were dashing. This way, they'll continue moving in the dash direction after it finishes, instead of coming to an instant stop. If they're not holding any movement keys anymore, the momentum will fade away smoothly as it always does.

Dashing Variables

Let's set up our variables to begin implementing the dash. Write these variables **beneath your existing variables in the Player script**:

```
//Dashing
[Header("Dashing")]
[Tooltip("Total number of units traveled when performing a dash.")]
public float dashDistance = 17;

[Tooltip("Time taken to perform a dash, in seconds.")]
public float dashTime = .26f;
```

```
private bool IsDashing
{
    get
    {
        return (Time.time < dashBeginTime + dashTime);
    }
}

private Vector3 dashDirection;
private float dashBeginTime = Mathf.NegativeInfinity;
```

As you can see, the only variables we need to expose in the Inspector are the dash distance and dash time. The dash will travel in the movement direction by “dashDistance” units over “dashTime” seconds.

Beneath that, we declare a bool private property “IsDashing”. It only has a getter, and returns true if we’re currently dashing and false if not.

It uses the variable “dashBeginTime” declared further below, coupled with the dashTime, to know the time at which the last dash began and how long a dash takes. Comparing that to the current time, we can figure if we’re currently dashing or not.

The `Mathf.NegativeInfinity` reference we use when declaring “dashBeginTime” is a built-in shortcut to get the very lowest number we could possibly get.

This is just a safety measure: if we merely set the “dashBeginTime” to 0 at the start of the game, then we would technically be performing a dash as soon as the game begins. By setting the begin time to a super-low value, we ensure that even if we (for some reason) have an extremely long-lasting dash, we won’t be dashing right when the game starts.

In reality, we could probably get away with something like -5, because when would we ever have a dash that lasts five seconds or longer? But better safe than sorry, right?

For the `IsDashing` property, you might’ve instead thought to declare a normal bool variable and set it to true when a dash begins and then back to false when the dash ends. But our property is a little cleaner: the status of dashing is handled somewhat automatically. All we have to do to begin a dash is set the `dashBeginTime`, and the rest handles itself. `Time.time` will raise above “dashBeginTime + dashTime” on its own, and the property will begin returning false again.

Of course, we also need to know what direction the dash occurred in, so we keep a `Vector3` for that.

As a direction, this will be set to 0, 1, or -1 on its X and Z axes. When we move, we'll multiply the movement per second by this direction. For example, if we held only the D or right arrow key, the `dashDirection` would be (1, 0, 0). Multiplying that by a float value is just a simple way to apply the movement only to the appropriate axis: the other two axes are 0 and, thus, no movement occurs.

If we held D and W, we'd get (1, 0, 1) instead, applying movement also to the Z axis, and so on for all the different directions one can move with the WASD/arrow keys.

Dashing Method

The Dashing method will be much like our Movement method. It will handle all of the dashing-related logic, keeping it tucked away in a separate private method that gets called in our Update method. We'll declare it just below the Movement method we made before.

The logic will be fairly simple:

- If we are not dashing, we'll check if the space key is pressed. If so...
 - Figure out the dash direction by checking which movement keys are held. If no movement key is being held, we don't perform the dash at all. If at least one key is held, we perform the dash.
- If we are dashing, we'll simply move along the dash direction using the `CharacterController`, just like when we move normally. Since the `IsDashing` property will automatically start returning false as soon as the dash time is up, we don't have to worry about bringing the dash to an end.

To perform the dash, we'll need to set `dashDirection` to the movement direction held by the movement keys. Knowing that, we'll be sure to store the movement direction in a variable as we go, so we don't have to check the input twice. We'll also need to set `dashBeginTime` to the current `Time.time`. That will cause the `IsDashing` property to begin returning true, which prevents us from performing a dash again while we're already dashing.

On top of that, we'll set our `movementVelocity` to full speed in the dash direction. This way, when the dash finishes and normal movement takes control instead, we'll have velocity in the dash direction.

We'll also rotate the model Transform to face along the dash direction.

Let's see the code – remember, place this code beneath your Movement method **in the Player script**:

```
private void Dashing()
{
    if (!IsDashing) //If we aren't dashing right now
    {
        //If the space key is pressed
        if (Input.GetKey(KeyCode.Space))
        {
            //Find the direction we're holding with the movement keys:
            Vector3 movementDir = Vector3.zero;

            //If holding W or up arrow, set z to 1:
            if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow))
                movementDir.z = 1;

            //Else if holding S or down arrow, set z to -1:
            else if (Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.
                DownArrow))
                movementDir.z = -1;

            //If holding D or right arrow, set x to 1:
            if (Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.
                RightArrow))
                movementDir.x = 1;

            //Else if holding A or left arrow, set x to -1:
            else if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.
                LeftArrow))
                movementDir.x = -1;

            //If at least one movement key was held:
            if (movementDir.x != 0 || movementDir.z != 0)
            {
                //Start dashing:
                dashDirection = movementDir;
            }
        }
    }
}
```

```

        dashBeginTime = Time.time;
        movementVelocity = dashDirection * movespeed;
        modelTrans.forward = dashDirection;
    }
}
else //If we are dashing
{
    characterController.Move(dashDirection * (dashDistance / dashTime)
        * Time.deltaTime);
}
}

```

Everything here is pretty self-explanatory, except perhaps the “modelTrans.forward = dashDirection;” line.

This .forward property is a Transform member that allows us to get or set the forward-facing direction of the Transform. We’ve used it before to get the direction the Transform is facing, but never to set it. It works just as you’d expect – if given a direction you want the Transform to face along, you can set its .forward to instantly turn it so that its front side faces that direction.

Similar variants exist for .right and .up to set the right-side facing and the top-side facing of a Transform. Conversely, if you wanted to point the bottom, back, or left side along a specific direction, you’d simply set the .up, .forward, or .right, but flip the direction you’re applying to the value by multiplying it by -1. That is to say, if you wanted to point the left side along a direction, all you have to do is point the right side at the opposite direction, right? That’s why there are only .forward, .right, and .up members and not opposite members like .back, .left, and .down.

The way we move our CharacterController is much like with movement. In order to achieve the desired distance, we need to move by the dashDistance divided over the dashTime and, of course, multiply by Time.deltaTime to ensure that it happens per second.

Final Touches

We need only make a few more changes to fully implement dashing.

First, we need to go to our “private void Movement()” method and wrap all of that movement code in an “if (!IsDashing)” block:

```
//Only move if we aren't dashing:
if (!IsDashing)
{
    //...the rest of the movement code goes here
}
```

This prevents any of the movement logic from occurring while a dash performs.

We also need to add the Dashing method call in Update, so that the Update method looks like this:

```
void Update()
{
    Movement();

    Dashing();
}
```

You’d be surprised how easy it is to forget this step, test the code, and wonder why nothing is happening!

One final bit of robustness we’ll add is to ensure that dashing always ends as soon as we die. That way, if we’ve set up our game to have a very low respawn wait time and a long-lasting dash, the dash won’t continue after we respawn.

In the **public void Die()** method, add this line somewhere within the if block:

```
dashBeginTime = Mathf.NegativeInfinity;
```

Again, we apply it to `Mathf.NegativeInfinity` instead of 0 just to be extra safe.

That should do us. You can now test your dashing in-game: just hold any movement keys down (at least one key is necessary) and press Space.

If you want, you can play with the dash distance and time until you like the way they feel. I will teach you one final trick when dealing with “distance over time” like this. If you want to get the actual “distance per second” of the dash so you can better compare the

dash speed to the regular movespeed, simply divide `dashDistance` by `dashTime`.

For example, with the default settings given to the script, we have a `dashDistance` of 17 units and a `dashTime` of .26 seconds.

With that, we have $17 / .26$. This means the dash will move the player by a little over 65 units per second. Our normal movespeed, by default, is 24 units per second.

When trying to balance the speed, this can be a useful way to look at it.

Dash Cooldown

One final implementation we might want is a dash cooldown. If you're into making games, I'm sure you know that a **cooldown** just means a duration one must wait after doing something, before they're allowed to do it again. If we don't have one, the player might as well be dashing all the time, right? We want to make them covet their dash a little bit more than that.

First, let's add a variable for the dash cooldown and a property "CanDashNow". Of course, we'll **put these with the rest of the dashing-related variables**:

```
[Tooltip("Time after dashing finishes before it can be performed again.")]
public float dashCooldown = 1.8f;

private bool CanDashNow
{
    get
    {
        return (Time.time > dashBeginTime + dashTime + dashCooldown);
    }
}
```

The cooldown is just a standard float that we can set in the Inspector. The `CanDashNow` property is similar to the `IsDashing` property. It can only be gotten, not set. It returns true if the current time has gone past the dash begin time, plus the time it takes to perform a dash, plus the dash cooldown wait time.

Next, **in our `Dashing()` method**, we'll change the first "if" statement from this

```
if (!IsDashing) //If we aren't dashing right now
```

to this:

```
//If we aren't dashing right now, and dash is not on cooldown:
if (!IsDashing && CanDashNow)
```

This will cause a sneaky problem to occur. One of those “I don’t throw an error, but I fundamentally break your game” sort of problems. Play it now, and the dash will last as long as the cooldown + the wait time together.

This is because we changed our “if”, but we’re still performing the dash movement in an “else” below it.

Now that the condition of the “if” has changed, the “else” doesn’t quite cover the same situation anymore.

Before, we just said “Are we not dashing?”

So the else says “Are we dashing?”

Now, we say “Are we not dashing, and can we dash now?”

This makes the else say “Are we dashing, OR can we NOT dash now?”

Of course, as soon as the dash begins, we can’t dash until the cooldown is up. So the “else” will keep happening until the dash goes off cooldown!

There are two ways to solve this.

The first way would be to change the “else” to a separate “if” that checks IsDashing itself, by replacing the “else” line with this and leaving the following code block the same:

```
if (IsDashing)
```

The second way is to keep the first “if” and the “else” as is and then add the CanDashNow check into the same “if” that checks for the space key being pressed instead. In other words, leave the “else” as it is, and change the first two “if”s to this instead:

```
//If we aren't dashing right now:
if (!IsDashing)
{
    //If dash is not on cooldown, and the space key is pressed:
    if (CanDashNow && Input.GetKey(KeyCode.Space))
    {
        //[...]
    }
}
```

This way, the “if” that corresponds to the “else” remains the same, so the logic isn’t broken in the same way it was before.

Either way works. One might argue that the second way is “more efficient” or “cleaner.” One might also argue that the dash occurs as soon as the key is pressed in the first way, which can make it feel very slightly more responsive.

This is because the “else” won’t occur until the next frame; as long as the corresponding “if” evaluates to true, the “else” is guaranteed not to happen on that frame. If the “else” was changed to an “if” block instead (the first way), it would evaluate to “true” as soon as the dash begins, on that same frame. If the player is experiencing choppy framerate, this could make a more noticeable difference, since there will be more time between frames. However, it could also make the dash distance a little greater than you might expect in such situations, since the player gets that extra frame of movement.

It’s interesting to see these little differences that come with each implementation, and in some situations, they might count. Here, it really isn’t going to be a big deal either way – just little nuances.

Summary

This chapter gave our player an extra tool to use in avoiding obstacles – a quick dash. Our player’s movement for this project is now at its final stage, with no more features to add.

We also learned how the Transform facing direction properties “.up”, “.right”, and “.forward” can be used to turn a certain side of the Transform toward a direction, and we learned how to pinpoint some tricky changes in logic that can occur in your “else” blocks when you make changes to their corresponding “if”.

CHAPTER 21

Designing Levels

In this chapter, we'll start thinking about how to support multiple levels that the player can choose from when they first load the game. We'll make separate scenes for each level and position a camera in each one that views the level from above as a sort of "sneak peek" when choosing levels.

Prefabs and Variants

Before getting into designing levels for your game, you ought to consider how you'll go about making little tweaks in the balance as you go along. We discussed the purpose of prefabs and their variants early on. This is one of those moments where you'll want to give them some thought – it might save you some heartache down the road.

We can always perform overrides on prefab instances to make, for example, superfast patrollers, shooters that fire more or less often, or projectiles that move faster or slower or are bigger or smaller.

However, if at any point you need to make a change, you'll end up with scattered overrides across various instances and scenes.

A smarter alternative might be to make prefab variants for the slightly different versions of your obstacles. If you stick to this, you can have consistent variations across your levels that the player will recognize. Make a fast and slow variant for the projectile prefab. Make a slow, large patroller variant. Make a quick, tiny wanderer variant.

This not only regulates your game design to make it easier for the player to predict and adapt to situations, but it gives you a clean setup that can be changed across all your levels just by tweaking the prefabs and/or their variants. If you overrode 50 different shooter instances across 15 different levels, you couldn't change them all at once if you ever decided to make the player move a little faster or slower and found that your shooters now needed to be tweaked in comparison.

Part of the fun of making games is playing with the numbers and tweaking things until they’re just right, so I’m not going to tell you how to design the game yourself. That’s not really in the scope of a book about game programming, after all. I will, however, teach you how to create a simple variant of a Shooter that fires faster than the normal prefab.

In case you forgot, a prefab variant is created by right-clicking a prefab asset in the Project window, unfolding the “Create” menu, and then selecting “Prefab Variant,” as shown in Figure 21-1.

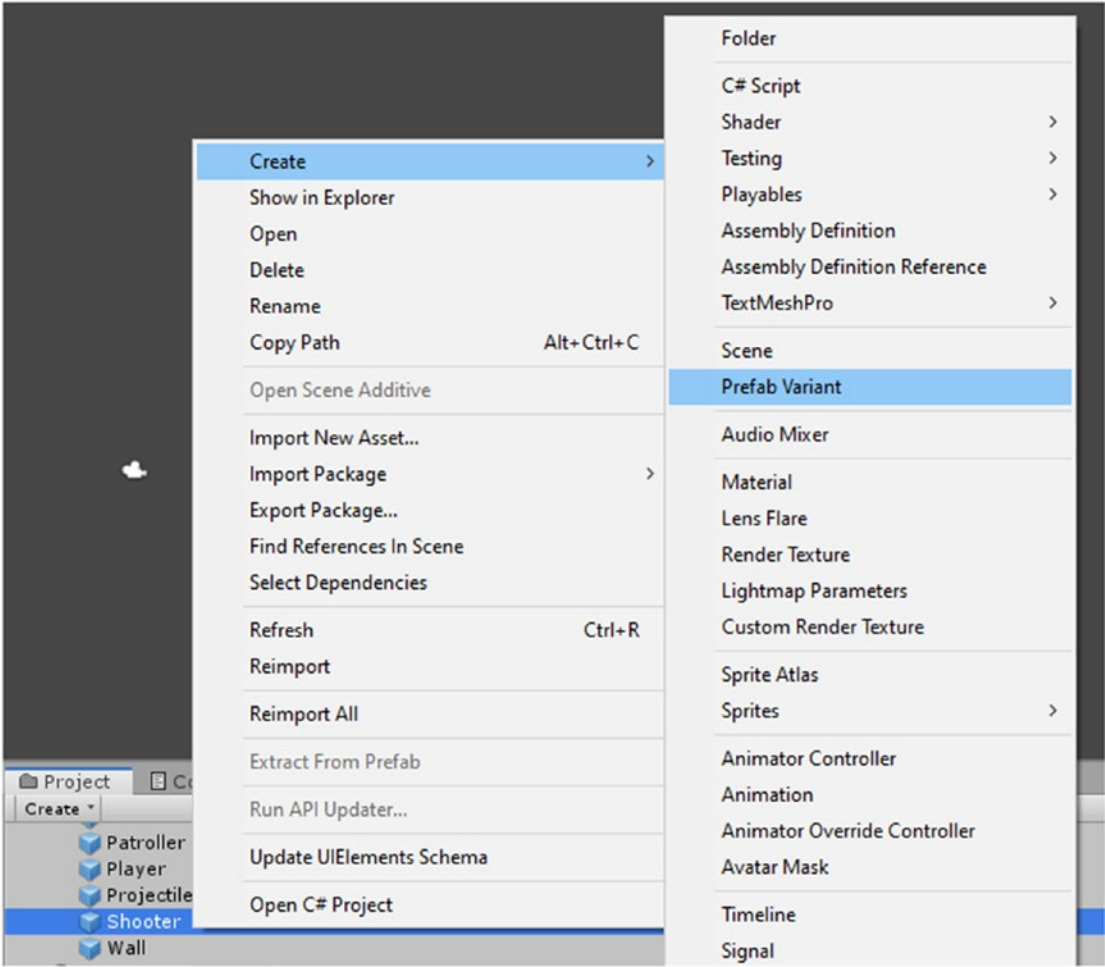


Figure 21-1. The right-click context menu leading us to create a prefab variant for our Shooter prefab

This will create a new asset acting as a variant to the Shooter prefab, and you can type whatever name you want for it. Let's name it "Shooter (Fast)". If you're going to have variants for many of your obstacle prefabs, you may want to consider ways to ensure that your assets remain tidy. For example, by naming it "Shooter (Fast)" instead of "Fast Shooter", we ensure that the variant remains next to the original Shooter in the Project view – since the names are sorted alphabetically, we want a name that's similar enough to the base prefab to keep both assets sorted together. If you'd rather, you could also put all prefabs and variants for each type of obstacle in a dedicated folder, such as "Shooters" or "Patrollers."

The variant works like a prefab, so you can open it by double-clicking it in the Project view if you need to make edits to child GameObjects. We just need to change how frequently our shooter fires, so we can just select the "Shooter (Fast)" in the Project and use the Inspector to change its Fire Rate to a lower value, like .5.

After that, you can drag and drop the "Shooter (Fast)" into the Scene and test it out. If you want visual indication of which shooters are faster, you can use a separate material for them. Remember to open up the variant asset and apply the material to it there, rather than applying it only to the instance you've placed in the scene.

You don't have to stop there if you don't want to. Here are some ideas on further ways to use variants:

- Projectiles that move faster or slower and are larger or smaller. You can then create Shooter variants that use different types of projectiles.
- Wanderers that are smaller, but retarget more frequently, move quicker, and don't take as long to start moving after retargeting.
- Larger or smaller patrollers with varying movespeed.
- Shooters that spin in circles. Remember that Rotating script we made before this project? Copy that over, throw it on a shooter, and have it spin while it fires projectiles.

Making Levels

I'm not going to try to guide you through the creation of a full level and all of its obstacles, because that would be a lot of tedious work for you, and it probably wouldn't be very fun for you to spend time recreating my idea of a level anyway. You have the tools

to make your own levels now, so I'll leave you to it. Instead, I'll give you some tips on how to start, and we'll start thinking about what needs to go in each level to make the whole process come together.

Before we begin, let's create a material for the floor Plane we've been using. It's a somewhat stale gray by default, and with the way our camera is positioned, it's filling any space that isn't covered by walls, obstacles, or the player. Let's make it a bit more appealing. Create a material named Floor and apply it to your Floor GameObject (or Plane if you never renamed it). I find that a dark-blue color with a hex value of 1D2A36 works well with the color of our walls, and the darkness makes the pale yellow of our Player stand out.

Moving on, let's make a new scene. To do so, just use the Ctrl+N hotkey (Cmd+N for Mac users), or navigate to File in the top-left corner of Unity and select New Scene.

Once you've made the new scene, use the Hierarchy to select and delete the Main Camera that comes with it by default.

First, a level should always have

- A Plane positioned at the world origin (0, 0, 0) and scaled on the X and Z axes so it's large enough to cover the whole screen at all times. Just go crazy and give it 1000 scale on both axes if you want. And don't forget to apply your Floor material to it.
- A Goal prefab instance somewhere the player can reach it. What's a level you can't win, after all? If you never made a prefab out of your Goal, you can go back to your "main" scene and make one real quick. If you've deleted your Goal and never made a prefab, you'll have to recreate it (look over Chapter 17 again).
- A Directional Light to make sure there's a global source of light for the scene. We've just been using the default light that comes with a new scene, but you can change the settings of the Light component in the Inspector for the Directional Light if you want to play around with it. The most notable settings would be the color of the light (a pale yellow by default, to somewhat mimic the sun) and the intensity value, which determines how brightly the light shines.
- A Player prefab instance at the location you want the player to spawn. You can start it out at position (0, 0, 0) and move it if you ever find a need to.

If your scene looks too dark, it could be due to the “Auto Generate Lighting” option being off by default in new scenes.

Check all the way at the bottom-right corner of the Unity editor. Does it say “Auto Generate Lighting Off” as shown in Figure 21-2?

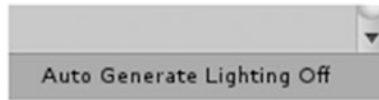


Figure 21-2. The Auto Generate Lighting option is shown in the bottom-right corner of the Unity editor. Here, it is off

You can click that text to bring up the Lighting window. The bottom of this window shows a section containing a checkbox with the text “Auto Generate” next to it. Check that box, as shown in Figure 21-3, and the scene should begin looking properly lit.

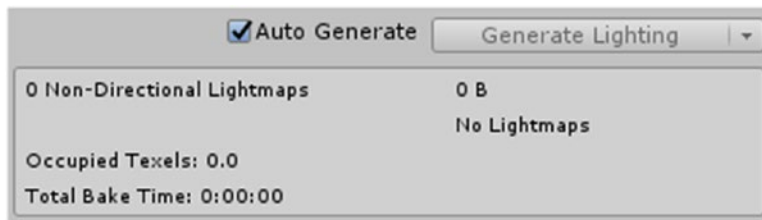


Figure 21-3. The bottom of the Lighting window, where the Auto Generate checkbox has been ticked

Adding Walls

With that sorted, you can place an instance of your Wall prefab down to begin blocking out the level so the player can’t just go wherever they want. Once you place it, move it up so its bottom is aligned with the floor correctly. You can do this by setting its Y position to half of its Y scale.

So you don’t have to fix the Y position every time you want a new wall, just copy-paste that wall from here on out.

As we mentioned before, the rect transform tool (hotkey T) can be quite useful for moving and sizing these walls. The tool tends to work best when viewing the walls from above. Remember, you can use the gizmo in the top-right corner of the Scene view to quickly assume a top-down view on the level – just click the green cone.

As you add walls, you can fill the space you block out with obstacles. Use your imagination and come up with some trials for the player. Or don't – you don't have to be a level designer to be a programmer, after all.

Level View Camera

Next, we're going to make a means of previewing the level before playing it. This will be done with a simple Camera GameObject. We'll set it up to view the level from whatever vantage point we agree with for our preview. In the following chapter, we'll make use of this camera when the player is viewing the level before deciding if they want to play it or not.

If you haven't already, delete the preexisting camera in the scene and make a new one. Rename it to "Level View Camera" (without the quotes). Make sure you get the name exactly right, because we're going to be finding it by name with our code later!

To easily position the camera how you like, just move your Scene camera to view the level from a nice angle by holding right-click and using the WASD keys. Then, select the Camera in the Hierarchy and, using the top menu, select **GameObject ► Align with View**, or use the hotkey **Ctrl+Shift+F**. This will place the Level View Camera right where your Scene view camera is currently positioned and even point it in the same direction too. This beats dragging and rotating the camera around with the transform tools.

Summary

This chapter shed some light on the process of designing levels by adding walls, using our prefabs, and creating variants to spice up the gameplay and how a new scene should be set up for a level.

Every level should have the following:

- A plane for the floor
- A camera named "Level View Camera" positioned where you want the level to be previewed from
- A Directional Light (the default one will do)

- A Player prefab instance where you want the player to start the level
- A Goal prefab instance wherever you like

In the next chapter, we'll set up the flow from one scene to another so we can preview levels and properly implement menus that let the player get to all of the different levels we make. If you're having fun designing levels, don't let all this talk about progress stop you, though – knock yourself out.

CHAPTER 22

Menus and UI

Unity has a feature-packed UI management system that was introduced more recently, where you mostly design your UI through GameObjects with relevant Components on them. But before this, we had an old code-based system for drawing UI to the screen primarily through method calls within our scripts.

Our next example project will dive into more detail on using this new UI system, because it needs UI as part of the core game experience. For this game, we're just trying to create a rough and dirty system to allow players to load levels through a main menu. As such, I want to demonstrate how to use that old code-based UI – what we'd call the “legacy GUI” – to draw the menus quick and easy.

This can come in handy when you want a quick solution for GUI to test some features. It may not look pretty, but in the early stages of development, we often don't care about that so much. The legacy GUI is unlikely to be your best bet if you want to create a “real game,” but it's very simple to use and easy to integrate into your code, so long as you don't expect it to be fancy.

Scene Flow

Let's examine a quick overview of how the loading of scenes is expected to flow for our game by the time we're done with this.

The first scene in our build settings – the one that loads first – will be the main menu scene. We won't have anything in this scene but a basic camera and an empty GameObject with a script that handles the main menu GUI.

This script will be called `LevelSelectUI`. It will call a little built-in method to instruct Unity that its GameObject should not be destroyed when a new scene is loaded. This way, the `LevelSelectUI` script can keep running after a level scene is opened.

Each level will have a scene all to itself. They'll be numbered by their index in the Build Settings, and they'll only show in the menu once we've added them to the build settings.

An instance of the Player prefab will be in each level scene, but the Player script will be disabled by default by unchecking the box beside its name in the Inspector, as we learned before. As well as this, the Camera GameObject inside the Player will be inactive – not just the Camera component, but the whole GameObject.

You can make this change through the Player prefab. First, make sure you’ve updated the Player prefab with any overrides that aren’t applied yet. Do this by selecting the Player in your Hierarchy, then clicking the Overrides dropdown in the header of the Inspector, and selecting “Apply All.”

Now open the Player prefab by double-clicking it in the Project window. Make the camera inactive by default, and disable the Player script by default. Now any level scenes you created (assuming you made one in the last chapter) will update to have the player at the initial state we desire.

Each level scene will have a Level View Camera on it, as we described in the previous chapter. This camera will be enabled by default, so when the LevelSelectUI loads the scene, we see the preview of the level, not the player’s view of the level. The player model will show in the level, but since the Player script is disabled, we won’t be able to move or act.

As long as we’re not in the main scene, our UI script will draw us a button we can press to play the game.

Once that button is pressed, we disable the Level View Camera, enable the Player script, and activate the player’s camera. Now the level is being played officially. We don’t need the UI anymore, so we destroy the GameObject holding the LevelSelectUI script.

When the player wins the level or uses the escape menu to quit (we’ll implement that in the next chapter), they’ll be brought back to the main scene again. Since we’ll be loading the main scene again, the same GameObject with the LevelSelectUI script on it will be there waiting for us again, so we can select our next level to play.

To set this up, we’ll start by cleaning any excess stuff out of the “main” scene we’ve had around since the start. If you’ve made something of a level in that scene already and don’t want to lose it, just rename the scene in the Project window and then create a new scene named Main. Otherwise, if you just have a bunch of junk lying around for testing, make sure you’ve made a prefab out of anything important and applied any important overrides to existing prefabs; then delete it all away.

Create a script named LevelSelectUI and attach it to an empty GameObject named Level Selection in the scene.

Now go to your Build Settings and make sure the Main scene is at build index 0 and any levels you’ve created are beneath it, as shown in Figure 22-1.



Figure 22-1. The Build Settings with our Main scene located at build index 0 (listed on the right side), with a few level scenes beneath it

One final step you can take is to stop the camera in the main scene from drawing anything but solid color. In the Inspector for the Camera component, change the “Clear Flags” field – the first one listed for the component – from Skybox to Solid Color. The field directly beneath it, a color field titled Background, can be changed to whatever you like. The camera will draw that color in the background instead of drawing the default view of the sky. I’ll set mine to a gray blue with a hex value of 5B6980.

Level Selection Script

The first thing to do for our LevelSelectUI script is to make sure it doesn’t get destroyed when we load a new scene.

This is a simple method call away. Declare a Start method like this:

```
void Start()
{
    //Make sure this object persists when the scene changes:
    DontDestroyOnLoad(gameObject);
}
```

That’s all we need to do to handle that. Now when we change scenes, this object will stick around in the newly loaded scene. This also means that if we load the scene this object originated from, we’ll end up with two copies of it. That’s why we’ll destroy it when the player decides on a level to play: we don’t want it sticking around past that point.

Now, let’s get into the UI code.

To use the legacy UI, you have to declare a built-in method “void OnGUI()”. This method is called whenever a new GUI event happens. These events are a range of different things: the mouse moving, mouse buttons being pressed, or keys being pressed. Most often, the event that triggers an OnGUI call is simply a “Repaint” event that happens constantly, sometimes multiple times per frame, to update and draw the GUI.

The only time we can call GUI methods is within the OnGUI method.

These GUI methods can be accessed through two objects: GUI and GUILayout. They both have mostly the same functionality – the same methods going by the same names. The difference is that GUI requires that we specify a position on the screen and a width and height of each thing we want to draw, while GUILayout automatically lays itself out, determining the position and size of GUI elements on its own unless we override it.

Positioning elements on a screen with code can be a bit of a pain. We'll do it a little bit later, but for the most part, since we're making placeholder GUI anyway, we'll let the system lay it out for us to save some development time.

First, make sure LevelSelectUI has this using statement at the top of the script file:

```
using UnityEngine.SceneManagement;
```

We used this before to reload the scene when we made our Goal script. It gives us access to the SceneManager object, which we'll use when loading in new scenes.

Then declare some variables at the top of our LevelSelectUI script class:

```
//Build index of the currently-loaded scene.
private int currentScene = 0;

//Current scene's level view camera, if any.
private GameObject levelViewCamera;

//Current ongoing scene loading operation, if any.
private AsyncOperation currentLoadOperation;
```

The current scene build index (the number associated with it in the Build Settings) will be tracked so we can display the level number that we're currently viewing to the user in a text label with the GUI.

It starts at 0, because that's what we expect the main menu UI to be. When we load a new scene, we'll update this index to match it.

We'll also store a reference to the level view camera of the current scene, which will initially be null for the main scene, but will be set when we load a new scene.

The last variable is of a type we haven't used yet. An AsyncOperation resembles an ongoing operation that occurs asynchronously – meaning it happens while the game continues to play. An activity that's particularly laborious on the computer processor will cause the whole program to freeze until it finishes. This often makes for an unpleasant experience. In this situation, we can run the operation gradually in the background,

devoting only part of the processing power to it. This is what **asynchronous** means. The opposite, **synchronous**, is the standard way code runs: the processor will perform one activity at a time.

Loading scenes can cause a noticeable hiccup – it’s a somewhat demanding task. This is why there’s a way to load them asynchronously. But we often need to react when the scene finishes loading, so we need some way to track the progress of the operation. To handle this, the method that loads a scene asynchronously will return to us an instance of `AsyncOperation`. We can use this instance to check if the operation has finished yet. We’ll do so to know when we should try to find our level view camera. If we try to find it immediately after we begin loading the scene – even if we loaded the scene synchronously – it likely won’t exist yet.

Let’s get to the code that shows our basic GUI and allows us to load in new scenes. Declare a void `OnGUI` method in `LevelSelectUI`:

```
void OnGUI()
{
    GUILayout.Label("OBSTACLE COURSE");

    //If this isn't the main menu:
    if (currentScene != 0)
    {
        GUILayout.Label("Currently viewing Level " + currentScene);

        //Show a PLAY button:
        if (GUILayout.Button("PLAY"))
        {
            //If the button is clicked, start playing the level:
            PlayCurrentLevel();
        }
    }
    else //If this is the main menu
        GUILayout.Label("Select a level to preview it.");

    //Starting at scene build index 1, loop through the remaining scene
    indexes:
    for (int i = 1; i < SceneManager.sceneCountInBuildSettings; i++)
    {
```



```

//Show a button with text "Level [level number]"
if (GUILayout.Button("Level " + i))
{
    //If that button is pressed, and we aren't already waiting for
    a scene to load:
    if (currentLoadOperation == null)
    {
        //Start loading the level asynchronously:
        currentLoadOperation = SceneManager.LoadSceneAsync(i);

        //Set the current scene:
        currentScene = i;
    }
}
}
}

```

Since this is the `OnGUI` method, we can call `GUILayout` methods within it. Each one doesn't bother us about where on the screen we want the results to be drawn. It just moves them down as it goes, each one drawing underneath the last.

The first GUI method we see is a call to `GUILayout.Label`. A `Label` is simply a means of drawing some text on the screen. We draw the title of our game, "OBSTACLE COURSE".

We then react to the `currentScene` variable to display something different based on whether we're in the main menu or if we're already previewing a level.

The main menu is index 0, so anything that's not index 0 will be a level we're previewing.

If we're previewing a level, we show a `Label` telling the user which level they're viewing.

We then use a call to `GUILayout.Button`, wrapped in an "if" statement.

This method call not only shows a button on the screen but returns true if it was pressed on this event or false if it was not. Anything within that if block will be the code we want to run if the button is pressed. In our case, we run a method we'll be declaring in a moment: `PlayCurrentLevel`.

If we're not previewing a level, we must be at the main menu. We show a different `Label` instructing the user to click a level below.

Of course, we'll draw buttons to choose between the levels below. We'll do a for loop, starting at index 1 this time instead of 0. This loop will go over all of the

levels in the build settings, except for the one at index 0 (the main menu). To get the number of scenes in the build settings, we go through `SceneManager` to reference the `.sceneCountInBuildSettings` member.

For each level scene, we draw a button with the text “Level” plus the level number written on it.

If one of these buttons is clicked, and so long as we haven’t already started loading a scene, we begin loading the scene at the current index (“i”) with `SceneManager.LoadSceneAsync`. In our `Goal` script, we load the level synchronously, which is done with the method `LoadScene`. This time, we want to do it asynchronously, so we call `LoadSceneAsync` instead.

As we established before, it returns an `AsyncOperation`, which we apply to our `currentLoadOperation` variable to keep it around.

Once we determine that the scene has loaded, we’ll null out this variable so we can once again load a new level. Until that happens, our “if” prevents the user from loading a different scene while one is still processing.

Now we need some per-frame logic to detect when that operation finishes so we can do some setup.

We’ll do this with an `Update` method:

```
void Update()
{
    //If we have a current load operation and it's done:
    if (currentLoadOperation != null && currentLoadOperation.isDone)
    {
        //Null out the load operation:
        currentLoadOperation = null;

        //Find the level view camera in the scene:
        levelViewCamera = GameObject.Find("Level View Camera");

        //Log an error if we couldn't find the camera:
        if (levelViewCamera == null)
            Debug.LogError("No level view camera was found in the scene!");
    }
}
```

The `AsyncOperation` “`isDone`” member is a `bool` that we can use to check if it’s finished or not. So long as we have a `currentLoadOperation`, we’ll check if it’s done yet.

Once it's done, we set it to null again and use the `GameObject.Find` method to try to get a level view camera in the newly loaded scene.

`GameObject.Find` takes a string for a `GameObject` name and searches the scene for it. If it finds it, it returns it. Otherwise, it just returns null. One thing to note about this method is that it won't find `GameObjects` that are inactive.

If we did fail to find a level view camera, we will throw an error with `Debug.LogError`, which is like `Debug.Log`, except that it shows up as a shiny red error in the Console instead of a neutral message. This will alert us if we ever forgot to add the camera.

Now all we need to do is determine what happens when the player clicks the Play button.

We already wrote the method call, so let's declare the method itself:

```
private void PlayCurrentLevel()
{
    //Deactivate the level view camera:
    levelViewCamera.SetActive(false);

    //Try to find the Player GameObject:
    var playerGobj = GameObject.Find("Player");

    //Throw an error if we couldn't find it:
    if (playerGobj == null)
        Debug.LogError("Couldn't find a Player in the level!");
    else //If we did find the player:
    {
        //Get the Player script attached and enable it:
        var playerScript = playerGobj.GetComponent<Player>();
        playerScript.enabled = true;

        //Through the player script, access the camera GameObject and
        activate it:
        playerScript.cam.SetActive(true);

        //Destroy self; we'll come back when the main scene is loaded
        again:
        Destroy(this.gameObject);
    }
}
```

First, we deactivate the level view camera so it doesn't render anymore. We don't want two cameras trying to render at the same time.

Then we try to find the Player by name. There should be one in every level; if there isn't, we log an error. If there is, we grab the Player component from it and enable it.

We then reach through the Player script to access the "cam" member – which we'll need to declare because we haven't yet. Let's do that now, since our code will throw an error until we do. Open up your Player script. In the References header, add this variable:

```
public GameObject cam;
```

With that line added to the Player, your code shouldn't throw any errors.

This "cam" variable will be a pointer to the player's Camera GameObject. It'll be inactive by default, so we can't use `GameObject.Find` to get it. We'll just rely on a reference in the Player script, since we need to grab the script anyway to enable it. We still need to set the reference, though, so make sure you open the Player prefab through the Project and assign the reference there to apply it across all of your Player instances.

Anyway, after we've activated the camera, the only thing left to do is to bid farewell to the `LevelSelectUI` script and its containing GameObject by destroying them. Remember, if we don't do this, then it will still be drawing the GUI and it will still be persisting after the main scene is reloaded, leaving us with two separate instances of it, both drawing at the same time.

With this in effect, the system should now be fully contained. Starting in the main scene, play the game and you'll see the GUI in the top-left corner, as shown in Figure 22-2. Of course, it's not particularly fancy, and it's just a little thing up there, but it functions.



Figure 22-2. The main scene GUI, before any levels have been loaded. Here, we have two level scenes added to the Build Settings, providing us with buttons for Level 1 and Level 2

If you click a level button, the level should load in. If you've set up the level correctly, the Level View Camera should be active by default, while the player's camera is not. This means we see the level at first through the Level View Camera.

With a level loaded, the GUI will adapt to show a Play button. Pressing this will deactivate the level view camera, enable the player camera, and set the player into action, allowing us to control it.

Step on a Goal in the level, and you'll find yourself transported back to the main menu, where you can go through the same process again to play another level.

Summary

This chapter gives us some additional robustness for the game. We've given the player a menu they can use to select which level they wish to play and return them to this menu when they reach the goal of a level. We also switch the Player script off by default and provide a preview of the level, giving control to the Player script only after the user has clicked a Play button.

Some things to remember are as follows:

- To make a `GameObject` stick around even when we load a new scene in-game, call the **`DontDestroyOnLoad`** method and pass the `GameObject` as a parameter.
- **`OnGUI`** is a built-in event where we can call certain methods to draw GUI to the screen.
- **`GUI`** and **`GUILayout`** are two classes with static methods in them that we can call within `OnGUI`. They're mostly the same methods, but **`GUI`** methods must have their position and size specified in their parameters, while **`GUILayout`** methods will automatically position and size themselves.
- An **`asynchronous`** operation is one that occurs in the background, sometimes taking many frames to finish. A **`synchronous`** operation is one that occurs all at once, which can cause a drop in framerate (or even temporarily freeze the game) if the operation is particularly intensive to run.

- **SceneManager.LoadSceneAsync** loads a scene asynchronously, returning an **AsyncOperation** instance when it is called.
- The **AsyncOperation** has a “**isDone**” bool property that can be used to check if the scene has finished loading (true) or not (false).
- **SceneManager.sceneCountInBuildSettings** returns the number of scenes added to the build.
- The **GameObject.Find** method takes a string and attempts to find a **GameObject** with that name. If it finds one, it will return it. If not, it returns null.
- For one of our levels to work properly, it should have a Camera in it named Level View Camera. If the name isn’t exact, the **GameObject.Find** call won’t be able to find it.

Now all we need is some means of returning to the main menu from within the game. We’ll handle that in the next chapter.

CHAPTER 23

In-Game Pause Menu

Now we need some means of getting back to the main menu from within the game. We'll add this in the form of a menu that the player can open by pressing the Esc key. It also wouldn't hurt for that menu to pause the game while it's up and unpause when we close it.

Freezing Time

Everything in our game is based on time. We use `Time.deltaTime` to measure the distance something will move in a frame. We use `Time.time` to measure how long it's been since something happened. We use `Invoke` to time the player's respawning and the Wanderer retargeting.

This can all be manipulated with the `Time.timeScale` member.

It's a multiplier for how much time passes. It can be changed on a dime to make time pass slower, faster, or not at all.

By default, it's set to 1. If we wanted to make time pass at half speed, for example, we'd set it to .5. If we wanted time to pass twice as fast, we'd set it to 2. If we want to freeze time, we set it to 0.

The `timeScale` properly adjusts the timing of invoked methods, the `Time.deltaTime` variable, and the `Time.time` variable – which is great for us, because it means it just works with all of our existing features out of the box.

This means all we have to do to freeze time is this:

```
Time.timeScale = 0;
```

But we want to do it through a pause menu that's toggled with the Escape key.

The menu will show in the middle of the screen and have two options: one to resume the game and another to quit to the main menu – a simple, bare-bones menu.

To make sure it's only operable when the level has begun playing (not when we're still previewing), we'll go ahead and implement it on the `Player` script class. Since the

Player script is disabled until the game begins playing, we won't be able to open the pause menu when we're still in the level select menu.

First, declare a bool "paused" on the Player script:

```
private bool paused = false;
```

Then, declare a Pausing method, like our Movement and Dashing methods:

```
private void Pausing()
{
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        //Toggle pause status:
        paused = !paused;

        //If we're now paused, set timeScale to 0:
        if (paused)
            Time.timeScale = 0;

        //Otherwise if we're no longer paused, revert timeScale to 1:
        else
            Time.timeScale = 1;
    }
}
```

We'll call that in our Update method, and we'll also only run the Movement and Dashing logic if the game is not currently paused:

```
void Update()
{
    if (!paused)
    {
        Movement();

        Dashing();
    }

    Pausing();
}
```


Now we won't be able to detect input for dashes or movement when the game is still paused – just a little bonus robustness.

This will give us the pausing functionality, but no visual indication of it. We need an OnGUI method.

To handle the displaying of the box in the middle of the screen, we'll use a `GUILayout.BeginArea` call. This method lets us give it a `Rect`, which is short for rectangle, to define a box on the screen that we want to place `GUILayout` elements within.

It works somewhat intuitively. We call `GUILayout.BeginArea`, passing in the `Rect` we want to use as our area on the screen to contain the elements in. Then we run any `GUILayout` methods we want to put in the area – buttons, labels, whatever. Once we're finished, we call `GUILayout.EndArea` to break out of the area.

The part that's a bit tricky is creating the `Rect` to define the space on the screen that the area should take up.

A `Rect` constructor takes four values: an `X`, a `Y`, a width, and a height.

The `X` and `Y` work much like they do in world space units in our game, only they're resembling the 2D space of our screen now – and rather than working with the concept of units, each point is just 1 pixel of our screen. A pixel is one tiny colored dot that makes up your computer monitor. Most modern monitors are over a thousand pixels wide and tall.

Another notable difference is that a value of (0, 0) is not the center of the screen – it's the top-left corner. An increase in `X` goes toward the right edge of the screen, and an increase in `Y` goes toward the bottom edge of the screen.

Luckily for us, there's an easy way to measure the width and height of our screen in pixels: the `Screen.width` and `Screen.height` variables.

Cut those values in half, and we get the center of the screen: (`Screen.width * .5f`, `Screen.height * .5f`).

But the `X` and `Y` position of our `Rect` isn't the center of the area we're defining. That would make this too easy. Rather, it's the top-left corner of the area. That's generally how things work in 2D space – particularly with the legacy GUI systems.

Let's declare the method and see how we get around this problem:

```
void OnGUI()
{
    if (paused)
    {
        float boxWidth = Screen.width * .4f;
        float boxHeight = Screen.height * .4f;
```

```

GUILayout.BeginArea(new Rect(
    (Screen.width * .5f) - (boxWidth * .5f),
    (Screen.height * .5f) - (boxHeight * .5f),
    boxWidth,
    boxHeight));

if (GUILayout.Button("RESUME GAME",GUILayout.Height(boxHeight *
    .5f)))
{
    paused = false;
    Time.timeScale = 1;
}

if (GUILayout.Button("RETURN TO MAIN MENU",GUILayout.
    Height(boxHeight * .5f)))
{
    Time.timeScale = 1;
    SceneManager.LoadScene(0);
}

GUILayout.EndArea();
}
}

```

I've spaced out each individual parameter in the "new Rect" to make it easier to read. This doesn't affect the method call, it's just a formatting thing.

Again, the parameters, in order from top to bottom, are X, Y, width, height.

Before we start the area, we declare some shorthand local variables for the width and height of the box we want our area to resemble.

For the X and Y, we start at the center of the screen (width or height multiplied in half). But since it's the top-left corner of the box that we'll be placing at the center of the screen, it won't look right if we just do that. We need to shift it to the left by half of the box width and up by half of the box height. That's why we subtract half of the box width/height from the X/Y values, respectively.

After that, we can call our GUILayout.Button methods and then make sure we call GUILayout.EndArea to break the custom area we began earlier.

In those buttons methods, we have an extra parameter. The GUILayout.Height(...) calls coming after the button text are a way to customize a GUILayout method to have a

manually defined height. Rather than letting the system determine the height on its own, we take control of it by supplying this option.

A similar option exists for `GUILayout.Width`, among others, but we don't have any need for that: the buttons will size themselves to the width of their containing area automatically.

In our `GUILayout.Height` options, we specify that the height should be half of our `boxHeight`, so that the two buttons together take up the full height.

When the resume button is pressed, we unpause the game and revert the `timeScale` to 1 again so time flows as normal.

When the quit button is pressed, we load the main scene (index 0) and also make sure we reset the flow of time, since that variable won't reset itself when the scene reloads.

Oh, and remember that in order to access the `SceneManager`, we'll need to make sure the script has the correct using statement at the top:

```
using UnityEngine.SceneManagement;
```

And with that, we should now be able to play the game through the main scene, navigate to a level, and test our new pause menu.

Pressing `Escape` will bring the menu up, as shown in [Figure 23-1](#).

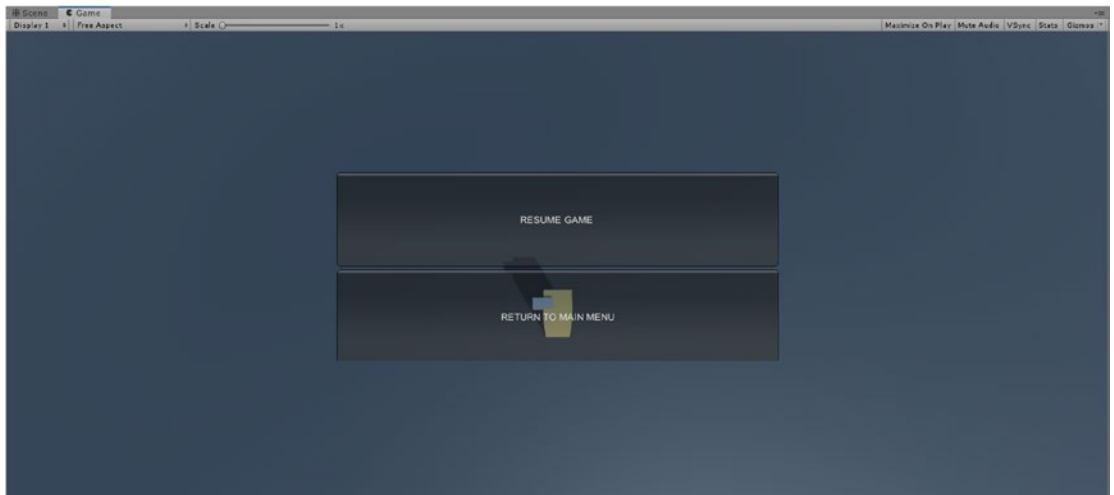


Figure 23-1. The in-game pause menu is shown over the player character

Returning to the main menu resets the process neatly, landing us back in the main scene where we can use the menu to go to a level again.

Summary

We've added one final component to our scene flow and menus: the option for the player to quit a level after they've started playing it with a menu opened by pressing the Esc key. This doubles as a pause feature by freezing time when the menu is open.

Some key points to remember are as follows:

- **Time.timeScale** is the multiplier for how much time actually passes in-game per second of real time. We can set it to change the rate that time flows. .5 is half as fast as normal, 2 is twice as fast, 0 freezes time, and so on.
- A **Rect** is a basic data type that resembles a rectangle: an X and Y position resembling the top-left corner of the rectangle and a width and height for the rectangle size.
- **GUILayout.BeginArea** is a method that starts a GUILayout area in a given rectangle of the screen, represented by a **Rect**. You call **BeginArea** and then any GUILayout methods you want to be part of the area, and then when you're done, you call **GUILayout.EndArea**.
- Most GUILayout methods that draw some element to the screen can be supplied options at the end of the call. We used **GUILayout.Height** to specify the height we wanted our buttons to use rather than allowing the layout system to figure it out on its own.

CHAPTER 24

Spike Traps

The final obstacle we'll implement will be a spike trap. It will teach you a new concept and give you more practice with the concepts we learned in previous chapters, like Lerp'ing over time, working with state, and Invoking methods to time transitions in the state.

Our spike trap will be a thin, square “plate” laid out on the floor which has lots of little “spikes” sticking out of it. For simplicity's sake, we'll make the whole thing out of cubes. When the trap activates, the spikes quickly raise, poking up out of the plate. They'll remain raised for a moment and then slowly lower back down.

While the spikes are raising, the trap is a Hazard, killing the player if they're standing over it. But when the spikes are raised or in the process of lowering back down, they act merely as a normal, physical collider that blocks the player from passing by, without actually killing them on touch.

Once the spikes finish lowering back down, the player can safely walk over the trap again. To accomplish this, we'll be using two separate Box Colliders on two separate GameObjects: one for the Hazard collider and one for the harmless collider. We'll activate and deactivate them as we go, based on the state of the trap.

Designing the Trap

Before we get into the little details on how we'll implement the trap in code, let's figure out how it will look and build it in the Scene. Once you're done, you should have a trap looking something like Figure 24-1.

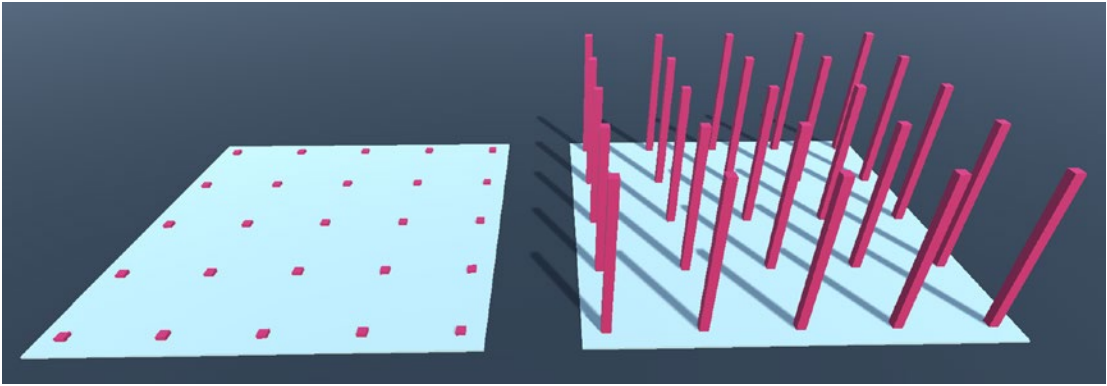


Figure 24-1. *A fully lowered Spike Trap (left) next to a fully raised Spike Trap (right)*

You can make the trap in any of your level scenes. Just move your camera off to a clear place where other stuff won't get in your way, and let's begin.

Create the root GameObject, an empty one named Spike Trap. Place it in the Hazard layer.

Right-click the Spike Trap and create a Cube child. Name it "Plate". Remove its default Box Collider. Unity should automatically place it in the Hazard layer to match its parent. If not, go ahead and do that yourself. Set its scale to (9, .1, 9), making it very thin, but wide and long. Set its local Y position to (0, .05, 0) to make sure the root GameObject (the pivot point) is at the bottom of it, not the center. There isn't much difference, but it's nice to be neat.

Now right-click the Spike Trap and add a new empty GameObject child. Name it "Spikes". This will be the parent of all the individual spikes.

Normally, you would have a model for the spikes, probably holding all of them in one mesh so a single GameObject could be used for all of the spikes at once. But we aren't creating our own meshes or artwork, so we'll use a separate GameObject for each individual spike. This will leave us with a somewhat cluttered Hierarchy, since we'll have lots of spikes in the trap, but we can always "fold up" the Spikes GameObject to tuck them away.

We'll use some tricks to distribute our spikes evenly across the surface of the plate without hand-placing each one. First, we'll need a single Spike instance that we can copy-paste around. Add a Cube as a child to the Spikes GameObject. Name it Spike and set its scale to (.2, 1, .2). The height is left at 1 to allow us to handle it through code – you'll see how we manage this when we get to that point. For now, just don't think about

how tall the spikes are. To align the bottoms of the spikes with the Spikes GameObject, set their Y position to .5. This is important. The bottom of each spike must be lined up with the Spikes GameObject – no higher or lower! A local Y position of .5 will do exactly that.

At this point, we're at a crossroads. How much do we care? Do we copy-paste our spikes and drag them around with the transform tools willy-nilly with no concern for their placement, or do we try to make them neatly lined up and pretty?

If you prefer a sloppier trap (it might look a bit more gnarly and brutal that way), you can do it that way – just make sure all of the spikes have the same Y position value.

But our trap, pictured in Figure 24-1, is made with some special tricks that distribute the spikes evenly without painstakingly dragging them into precise positions.

First, let's position this first spike in one of the corners. We'll put it at the top-left corner, with a little bit of margin between it and the edge of the trap. Set its position to (-4, .5, 4).

Now you have one spike, up there in the corner but not right at the edge of the corner, since that would look a bit odd. To copy-paste it and position each one an even distance away from the last without doing it by hand, we can use a little trick in the Inspector.

When you type a number value into a field in the Inspector, you can do math equations, and Unity will automatically calculate the result for you. For example, try typing $5 + 5$ into a number field, like a Transform's position. As soon as you press Enter or lose focus on the field by pressing Tab or clicking somewhere else, Unity will replace the equation with the result: 10.

We can use this to simply add to or subtract from one of the position axes of a Spike. Copy and paste the Spike instance and edit its X position field to add “+ 2” at the end. That is, leave whatever position value is currently in the field, and append the math at the end – if the field is -4, set it to “-4 + 2”. The result will be calculated and adjusted precisely, without the need for us to drag the spike with the mouse and try to get them all evenly spaced.

Do this until you run out of space to place more spikes. You should have a row of five spikes, stretching from the left side of the trap to the right side, as shown in Figure 24-2.

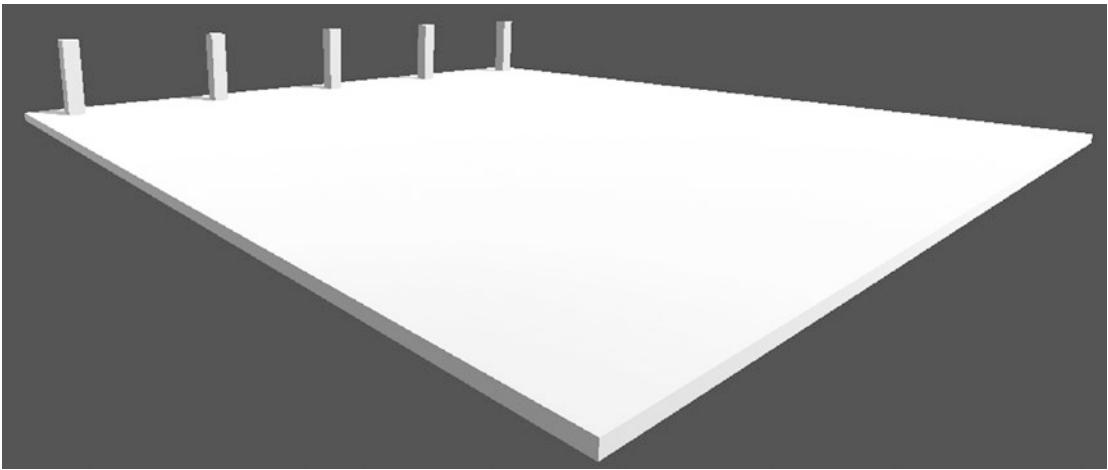


Figure 24-2. *The Spike Trap so far, with just one row of spikes*

We can't use the Inspector math trick on more than one GameObject at a time, though. So we can't just copy-paste a row of spikes and adjust their Z position to slide all 5 of them down at once.

Luckily, there's an easy solution. We can just make them all children of a temporary empty GameObject and then copy-paste and move the parent.

Create an empty GameObject in the Spikes holder, select all of the spikes, drag them over the new empty GameObject, and then copy-paste that GameObject. Then, adjust its Z position using the Inspector math trick, adding " - 2". You can then copy-paste that row and decrease its Z by 2 again. Repeat that until you have a total of five rows of spikes, covering the whole trap neatly.

We don't need those empty GameObjects to make the spike trap function, so if you don't want them around anymore, you can delete them. Of course, if we delete them now, it will delete the spikes inside them as well, so you'll first have to drag the spikes out to make them children of the Spikes holder once again. If you'd like to keep the spikes grouped by row, you can leave them as is, although you may wish to name them something descriptive like "Spike Row" instead of the default "GameObject" name.

And there you have it - precisely positioned spikes, without copy-pasting all 25 of them one at a time.

You can then create materials for the spike trap plate and the spikes themselves, if you want.

You can apply a material to all of the spikes at once by selecting them, navigating to their Mesh Renderer component in the Inspector, and finding the subheading

“Materials” within. It will show a field set to “Default-Material.” Just drag and drop your material from the Project window onto that field. Unity will detect that each Spike has the same component on it so that when we edit a field in that component, it applies the change to all of them. This is the same as dragging and dropping the material from the Project onto a GameObject in the Scene view. They both set the same field – doing it through the Scene is just a convenience feature.

While you’re at it, you can also remove the Box Collider component from all of the spikes at once. We don’t need colliders for individual spikes – we’ll use one collider that spreads over all of them. With all of the spikes selected, remove the Box Collider component in the Inspector.

For my spike trap, I’ve used a light-blue color with a hex value of DAFFF9 for the plate and a deep maroon with a hex value of F1236B for the spikes.

Raising and Lowering

You may be wondering why your spikes aren’t the same height as the spikes shown in Figure 24-1. This is because we haven’t set them up fully yet. There’s a new concept to be learned! Well, not so much a new concept as a side effect of an existing concept. We’re going to achieve the effect of raising/lowering our spikes with a bit of magic involving pivot points and scaling.

Like I said before, there’s a reason we made sure the bottom of each spike was vertically positioned to neatly line up with the Spikes GameObject.

Select your Spikes GameObject, switch to the scale transform tool (hotkey R), ensure that you are using the local pivot point (hotkey Z to toggle; the gizmos should appear at the base of the trap, not at the center of your spikes), and drag up the green box for the Y axis, increasing the Y scale. Since the spikes are all children of this empty GameObject, they all grow taller, as expected (see the left side of Figure 24-3). But the important thing is how the pivot point is scaling them “away from it.”

Try to instead select all of the Spike instances within the Spikes GameObject, but don’t select the Spikes GameObject itself. You can do this in the Hierarchy by clicking once on the topmost Spike, then holding Shift, and clicking once on the bottommost Spike. Scale up the Y axis again, just as you did before, and you’ll get something more like the right side of Figure 24-3. The spikes scale up by their individual pivot points – the center of each spike – and thus stick out further and further down as we scale them up.

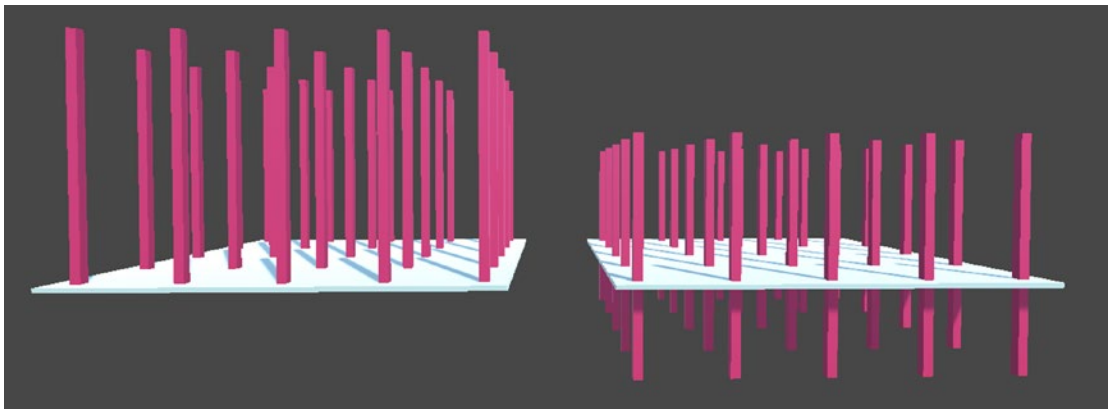


Figure 24-3. *Scaling up the Y axis of the Spikes GameObject to 5 (left) and scaling up the Y axis of each individual Spike to 5 (right). Both traps are at the same Y position*

The first option is what we want: the spikes should only go up, not down, when we raise the scale of the Spikes GameObject. That's why we've set our trap up in such a specific way. The spikes all have a Y scale of 1 and a Y position of .5. Since local position and scale are multiplied by the scale of the parent, this means they're exactly as many units tall as the Y scale of their pivot point – the Spikes GameObject – and they're exactly half that height above it, which keeps their bottoms precisely positioned at the pivot point. We can now define the height of all the spikes simply by setting the Y scale of the Spikes GameObject. We also need only change the scale of one GameObject to raise or lower the spikes, rather than keeping all of the spikes in an array, looping over them, and changing each one individually.

Writing the Script

Now you know how we plan on raising and lowering our spikes. Let's implement the part of the script that does this. We'll add the collisions after (don't worry, it's easy), so don't think about that part for now.

Create a script named `SpikeTrap`. Add an instance of it to the Spike Trap root GameObject we made.

First, let's go over the process the trap will be playing out. The trap will have four states, which we define in an enum named `State`, **declared in the `SpikeTrap` script class**:

```
private enum State
{
    Lowered,
    Lowering,
    Raising,
    Raised
}
```

The states are somewhat self-explanatory:

- While **Lowered** or **Raised**, the spikes aren't moving. They're sitting still, with the spikes either fully lowered or fully raised.
- While **Lowering**, the spikes are growing shorter, starting at the maximum height and lowering down to the minimum height.
- While **Raising**, the spikes are growing taller, starting at the minimum height and raising up to the maximum height.

Initially, it will have its spikes in the Lowered position, so that's what we set the state to when we declare the variable, **just below the enum declaration**:

```
private State state = State.Lowered;
```

"Lowered" is also the first item in the enum, so that's what it would naturally be set to if we didn't provide a value when we declared the state variable. Just to be explicit, we'll provide the default value anyway.

We'll also declare two const floats, depicting the height of the spikes when lowered and the height of the spikes when raised:

```
private const float SpikeHeight = 3.6f;
private const float LoweredSpikeHeight = .08f;
```

If you wanted to have different kinds of spike traps which have longer or shorter spikes – although it won't affect much – you could make these variables public instead of private and take out the "const" keyword, which will make the variables show in the Inspector so they can be customized.

Now let's declare the variables we'll be using, **down below the "state" variable**:

```
[Header("Stats")]
[Tooltip("Time in seconds after lowering the spikes before raising them
again.")]
public float interval = 2f;

[Tooltip("Time in seconds after raising the spikes before they start
lowering again.")]
public float raiseWaitTime = .3f;

[Tooltip("Time in seconds taken to fully lower the spikes.")]
public float lowerTime = .6f;

[Tooltip("Time in seconds taken to fully raise the spikes.")]
public float raiseTime = .08f;
```

The tooltips pretty much explain the purpose of each variable. The repeating process the trap will take is

- Begin raising over "raiseTime" seconds.
- Once raising finishes, wait "raiseWaitTime" seconds.
- Begin lowering over "lowerTime" seconds.
- Once lowering finishes, wait "interval" seconds.
- Repeat.

We'll also need a private variable that we'll use to track the Time.time that the trap started raising or lowering last:

```
private float lastSwitchTime = Mathf.NegativeInfinity;
```

And we'll need a reference to the Spikes GameObject, which we'll store in a variable named "spikeHolder":

```
[Header("References")]
[Tooltip("Reference to the parent of all the spikes.")]
public Transform spikeHolder;
```

Now we'll declare the logic which raises and lowers the spikes, **in the Update method**:

```
if (state == State.Lowering)
{
    //Get the spike holder local scale:
    Vector3 scale = spikeHolder.localScale;

    //Update the Y scale by lerp from max height to min height:
    scale.y = Mathf.Lerp(SpikeHeight, LoweredSpikeHeight, (Time.time -
    lastSwitchTime) / lowerTime);

    //Apply the updated scale to the spike holder:
    spikeHolder.localScale = scale;

    //If the spikes have finished lowering:
    if (scale.y == LoweredSpikeHeight)
    {
        //Update the state and Invoke the next raising in 'interval'
        seconds:
        Invoke("StartRaising", interval);
        state = State.Lowered;
    }
}
else if (state == State.Raising)
{
    //Get the spike holder local scale:
    Vector3 scale = spikeHolder.localScale;

    //Update the Y scale by lerp from min height to max height:
    scale.y = Mathf.Lerp(LoweredSpikeHeight, SpikeHeight, (Time.time -
    lastSwitchTime) / raiseTime);

    //Apply the updated scale to the spike holder:
    spikeHolder.localScale = scale;
```

```

//If the spikes have finished raising:
if (scale.y == SpikeHeight)
{
    //Update the state and Invoke the next lowering in 'raiseWaitTime'
    seconds:
    Invoke("StartLowering",raiseWaitTime);
    state = State.Raised;
}
}

```

The “if” and the “else if” both contain much the same code, with little changes to distinguish between raising and lowering. We use the `Mathf.Lerp` method, which is equivalent to Lerp'ing a `Vector3` or a `Quaternion`, only it works on two float values instead of vectors or Quaternions.

While Lowering, we Lerp from the fully raised height of the spikes down to the fully lowered height.

While Raising, we Lerp from the fully lowered height of the spikes up to the fully raised height.

As soon as the Lerp has brought the height to the exact height we seek, we change the state to Raised or Lowered, and then we use `Invoke` to kick off the next transition in the correct amount of time.

The fraction we pass to our Lerp calls is similar to what we did with the rotation of our Wanderers. When raising or lowering begins, we'll mark the `Time.time` that it began – that will happen in the `Invoke`d methods, which we'll be writing next. We use that to get the time, in seconds, since the transition first began. Using that, we can divide by the time we want the transition to take overall (`lowerTime` or `raiseTime`). This makes the fraction start at 0 and raise to 1 over the duration.

The `StartRaising` and `StartLowering` methods will be short and sweet. Declare them **somewhere above the Update method**:

```

void StartRaising()
{
    lastSwitchTime = Time.time;
    state = State.Raising;
}

```

```
void StartLowering()
{
    lastSwitchTime = Time.time;
    state = State.Lowering;
}
```

As you can see, these methods just update the `lastSwitchTime` so the `Lerp` knows when the transition began and set the `State` so the `Update` method will kick in and begin applying the transition.

With this, all we need to do now is ensure that the process gets kicked off in a `Start` method, **right above the `Update` method**:

```
void Start()
{
    //Spikes will be lowered by default.
    //We'll start raising them 'interval' seconds after Start.
    Invoke("StartRaising",interval);
}
```

Now, save that code and let's see it in action. It won't kill the player yet, but it should work visually. Before you test it, make sure you attach a `SpikeTrap` script component to the root Spike Trap `GameObject`, and make sure you set the "spikeHolder" reference to the "Spikes" pivot point `GameObject`. Also, the Spikes Y scale should be set to the same value as our `const LoweredSpikeHeight`, which we set to `.08`. That will ensure the spikes are in their proper, fully lowered position when the scene first loads in.

Adding Collisions

Now the trap works visually, but it still needs to interact with the player. This part won't be too hard to implement. As we said at the start of the chapter, the trap will be a `Hazard` that kills the player during the `Raising` state, and then while `Raised` and `Lowering`, it will act as a normal collider that blocks the player but does not kill.

Right-click the Spike Trap and add a new empty `GameObject` child. Name it "Hitbox". It should be in the `Hazard` layer. Add a `Hazard` script component and a `Box Collider`. Make sure the `Box Collider` is marked as a trigger collider and give it a `Size` vector to fit the width and length of the plate but make it a bit taller. Something like (9, 4, 9) will do.

This will cause it to stick through the ground, since the pivot is at its center. We can raise it up by changing the Center vector to (0, 2, 0).

Now, copy and paste the Hitbox. Rename this new one to “Collider”, remove the Hazard component, and make its collider **not** a trigger. This is the one that will physically impede the character, but not kill them on touch. Switch its layer to Default. It’s not a hazard, it’s pretty much equivalent to a wall.

By default, make both the Hitbox and Collider GameObjects be inactive by selecting them and unchecking the checkbox at the left side of their name in the Inspector. The default state of the Spike Trap is Lowered, so neither collider should be active at the start.

Let’s make it operational now. We just need to sprinkle some lines of code in here and there. All we’re doing is calling `GameObject.SetActive`, which we’ve used before. You’ll recall that it takes a single parameter, a bool that should be true to set the state to active or false to set the state to inactive.

Add these reference variables to the `SpikeTrap` script:

```
public GameObject hitboxGameObject;
public GameObject colliderGameObject;
```

Save, and drag and drop our Hitbox and Collider GameObjects onto their corresponding reference fields in the Inspector to set up the references.

Now, add this line to turn on the Hazard collider **in the StartRaising method**:

```
hitboxGameObject.SetActive(true);
```

Add this line to turn off the physical collisions once we’ve fully lowered, **after we've Invoked StartRaising in the Update method**:

```
colliderGameObject.SetActive(false);
```

Down below, **after we've Invoked the StartLowering call**, add these lines to both enable the physical collisions and disable the hitbox:

```
//Activate the collider to block the player:
colliderGameObject.SetActive(true);

//Deactivate the hitbox so it no longer kills the player:
hitboxGameObject.SetActive(false);
```


That'll do it. If you've set everything up correctly, the trap will now kill the player if they're standing on it when it's raising, and once it finishes raising, it will block the player until it fully lowers.

If you haven't already, don't forget to make a prefab for the Spike Trap. With all that spike setup, it would be extra painful to accidentally delete it with no way of getting it back!

Summary

This chapter gives us our final obstacle type, a deadly (if a bit blunt) spike trap that shows a unique behavior of toggling between a Hazard collider and a normal, nonlethal collider based on its state.

We've learned that when scaling a parent, the children scale relative to the pivot point. We've demonstrated how to introduce a pivot point to all of our spikes to not only scale them all at the same time but also to make them scale how we want them to. We also exercised some tricks to quickly set up our spikes by row instead of creating each one separately.

This marks the final implementation of our first example project.

CHAPTER 25

Obstacle Course Conclusion

At last, we've reached the end of our first project. Let's go over the final steps you might take when finished with a project: "building" it so that others can play it. I'll leave you with some further ideas for features to add to the game as well.

Building the Project

Although our project is anything but a polished and pretty gem waiting to be distributed to the masses, we can still get some hands-on experience with the building of a Unity project.

We discussed this topic briefly already. Building a project converts it to a format acceptable for an end user who wants to play your game. It copies our Unity project into a set of files that can be run without the Unity editor. This is necessary if you ever want to make a game available to a base of players – you very well can't ask them all to download the Unity editor and play your game through it, and you certainly don't want to give out all of your code and assets to the players.

Building a project is a simple task. It's mostly done with a few button presses. You've already seen the Build Settings menu, shown in Figure 25-1. It is found under File ► Build Settings or with the hotkey Ctrl+Shift+B.

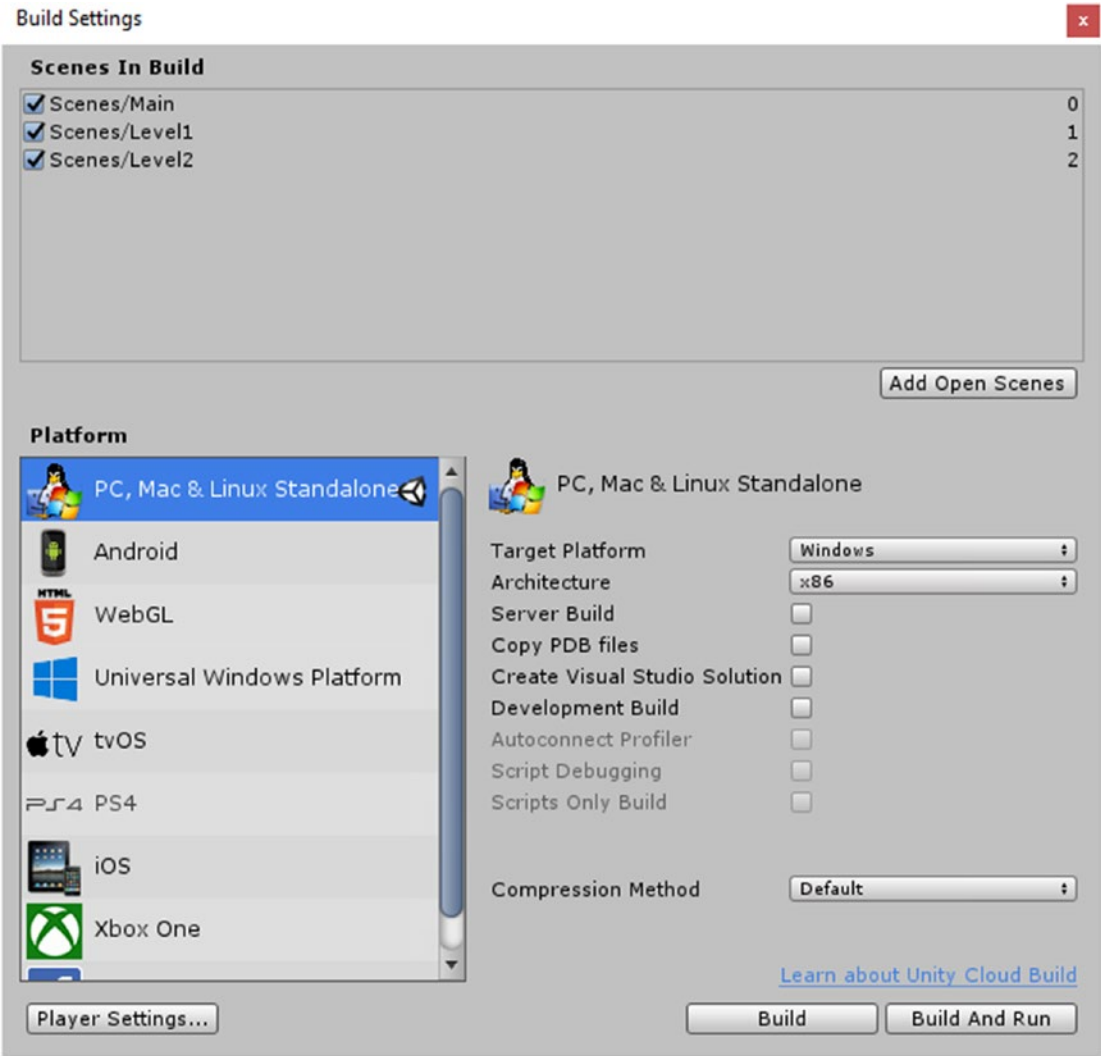


Figure 25-1. The Build Settings menu for our project, set to build for Windows

We previously learned about the top section, titled Scenes In Build. The section beneath it is where you pick the target platform to deploy to (left) and related settings (right). By default, it's PC, Mac & Linux Standalone. The project will be deployed to a particular operating system (Windows, Mac, or Linux). This can be changed with the Target Platform field, but support for building to each operating system comes from a different Build Support module installed through the Unity Hub. If you want to build to a target platform that isn't listed, you probably don't have the module installed. All you

have to do is go through the Unity Hub program and install the Build Support module for the operating system you want to target, which we talked about back in Chapter 1.

The remaining fields aren't particularly relevant to the everyday user. All you should need to do is just click one of the two buttons in the bottom-right corner: Build or Build And Run, the only difference being whether or not the built project will be automatically executed when it finishes. Before building can begin, you'll have to select a folder to store the built project in. Unity will generate the necessary files within that chosen folder.

Player Settings

The Player Settings can be accessed with the button in the bottom-left corner of the Build Settings window (shown in Figure 25-1). They can also be reached through Edit ► Project Settings and then by clicking the Player tab on the left.

Player Settings relate to the built program. There are fields for a variety of different settings, but most of it doesn't really concern your average hobbyist user, and when you hit the point that it does concern you, there are plenty of places to learn about it. The most important area to look at is the **Resolution and Presentation** section, displayed in Figure 25-2.

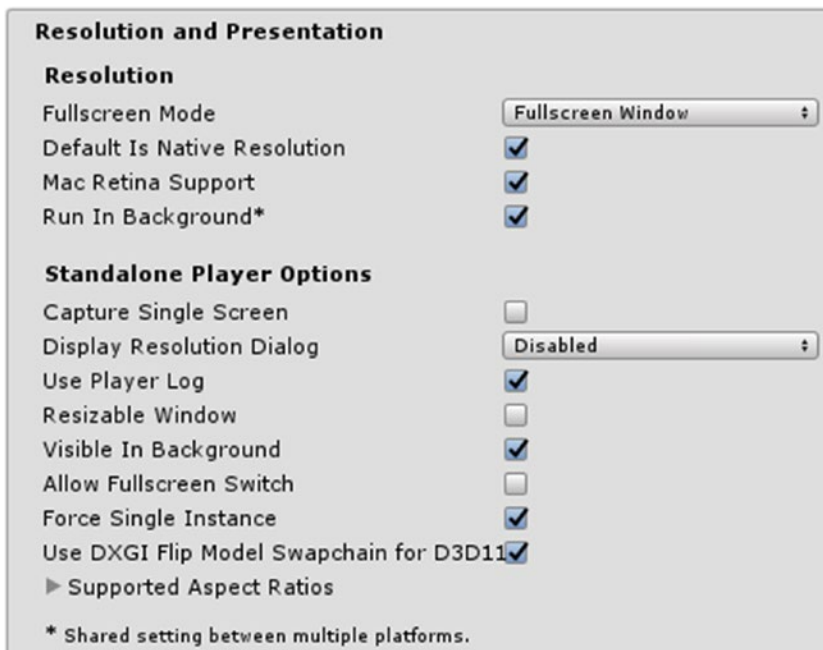


Figure 25-2. The Resolution and Presentation section of the Player Settings

Here's a quick overview of the fields that may interest you in this section:

- **Fullscreen Mode**

Defines the default display for the window. Is it fullscreen or windowed by default? If it's windowed, it will be a movable window that is not fullscreened (not a "windowed fullscreen").

- **Default Is Native Resolution**

If this box is checked, the game window will use the resolution of the user's computer by default. If you uncheck the box, two extra fields will appear, allowing you to set the default height and width of the game window yourself.

If you've set Fullscreen Mode to Windowed, this field won't show.

- **Run In Background**

Check this box if the game logic (your scripts, physics, etc.) should continue to operate even when the user minimizes the window or loses focus on it. Alternatively, if the box is unchecked, the game will pause (freeze in place) whenever the user minimizes it.

- **Display Resolution Dialog**

This setting defines whether or not Unity shows the resolution dialog when the game starts. This is a popup box which lets the user make their own decision for the resolution when they start the game.

By default, it is Disabled. It's recommended that you don't rely on this feature, since Unity has deprecated it – meaning they don't plan on supporting it in the future.

- **Resizable Window**

If this box is checked, the window can be resized by the user clicking and dragging on the corners or edges. If it's not checked, the window will be stuck in the default size you've specified.

- **Allow Fullscreen Switch**

If checked, the user can use the default hotkey for their operating system to toggle fullscreen on or off (in Windows, this is done with Alt+Enter).

- **Force Single Instance**

If checked, only one instance (window) of the game can be run at any time. Attempting to run a second one will simply bring focus to the already-running instance instead.

Aside from these features, there's also the option of defining a custom splash screen. The splash screen is a sequence of graphics that pop up when the game first runs, usually showing the logo of involved companies. You've probably seen this in games you've played before. By default, your game will have a splash screen of the Unity logo, lasting a couple of seconds. If you're using the free license of Unity, you cannot disable this – you don't get a state-of-the-art game engine for free without giving a little credit! You can, however, add your own logos to the splash screen, customize the background color, and add a background image. Of course, this is probably not of much concern to you at this stage – but do remember that, if you want to lose the official Unity splash screen when your game runs, you have to have a paid version of Unity.

Recap

This project has been a great, big series of firsts, each one making you more comfortable and equipped with programming in Unity. Before we move on to the next project, let's take a quick overview of what major concepts we learned to use in this one:

- Moving with a `CharacterController` component.
- Turning a number value from “per frame” to “per second.” If you want to move X units per second, move by $X * \text{Time.deltaTime}$ per frame.
- Slerping and Lerp-ing positions, rotations, and float values. This is pretty much just moving value A toward value B by a fraction of the distance between the two.

- Calculating the time since an event happened. When the event first occurs, set a variable X to the current value of `Time.time`. Later, use “`Time.time - X`” to get the seconds that have passed since.
- Invoking methods to run code after a specific duration has passed.
- Calling the `GameObject.SetActive` method to activate and deactivate `GameObjects` through code.
- Using a basic enum to depict the state of an object and changing the way they behave based on this state.
- Calling the `Random.Range` method to generate a random value between two given numbers.
- How a pivot point can affect the scaling of a `Transform`.

Additional Features

Now that you’ve got all this newfound experience, you can always add features to the project yourself. Thinking your way around a problem on your own is a great way to develop yourself as a programmer. Even if you bite off more than you can chew and attempt something too difficult, you’re probably going to pick up at least a few tidbits of information that make you more knowledgeable – and who knows, they might help you solve a different problem in the future.

Striking out on your own to implement a feature in a game project doesn’t mean you have to do it independently. A few Internet searches can make all the difference when it comes to solving problems or coding up new mechanics.

Getting an error message that doesn’t make sense to you? Use your favorite search engine, like Google, and type the error message out (you can even select an error in the Unity Console window and press `Ctrl+C` to copy the whole message). Chances are you’ll find other people making a similar mistake to the one you’ve made, asking others for help figuring it out, and there you may find your answer.

Unsure how to go about implementing something? Search for it, and so long as it’s not a particularly niche mechanic, you’ll likely be able to find a guide or tutorial for it.

Dealing with a component type you’re not used to? Search for it, and Unity’s documentation page will likely be one of your first results. You can see all of the members of any built-in class and read the descriptions for individual variables or

methods. Knowing what sort of relevant data and methods are at your disposal is often one of the first steps when it comes to planning your code.

In this day and age, the information you need is probably just a couple well-chosen search terms away.

If you ever run into trouble that you can't seem to think your way around and you can't find a solution online, you can either set it down and come back to it when you've gotten better (there's no shame in this!) or head to some online community and ask for help with your particular problem. Unity's official question-and-answer site, located at answers.unity3d.com, is a great place for beginners to present their questions or problems so more experienced Unity users and coders can help illuminate the correct path. If you're going to do this, be thorough. You don't want your question to generate more questions! Describe what you're trying to accomplish, what's in your scene, and what's happening that you're trying to fix. If code is involved, copy-paste any of it that relates so others can check it for errors – particularly the subtle ones you may not have noticed.

Sometimes, the act of trying to describe your problem with enough detail that someone else may be able to solve it for you is enough to help you realize what's going wrong. One day, you'll realize you've gotten so good that you don't need other people's help. You're good enough at finding the answer yourself that you solve the hurdles before you ever hit the point of asking for help. Or perhaps the problems you face will be deep enough that properly explaining them so others can help is more trouble than figuring it out yourself.

I'll give you some ideas for features you might want to add to this project on your own:

- **Teleportation**

Two teleporter pads are placed on the ground at different locations. Touch either one, and you'll get warped to the location of the other one. Use an Inspector reference to the Transform of the other teleporter pad to link them together. That way you can move them around – even in-game – and it'll still work.

- **Fancy shooters**

Make Shooters with multiple barrels sticking out in different directions, all of them firing at the same rate. Create shooters that are constantly spinning. Mix the two for much more daunting obstacles.

- **Checkpoints**

Larger levels can have checkpoints spread throughout them. When the player touches the checkpoint, they unlock it, making it the current checkpoint. Whenever they die, they respawn at the current checkpoint (if they have one unlocked). Just make sure the player can't go back and unlock a worse checkpoint by touching it again!

- **Lives**

Give the player a limited number of lives. Whenever they die, take a life away. If they run out of lives, kick them back to the main menu and make them feel bad by telling them they're a loser. This way, they can lose their progress earned by reaching new checkpoints. Consider also rewarding them with more lives when they reach a new checkpoint.

If you're tired of doing what I say, don't be afraid to try to implement something that you think would be fun or interesting. You might fail and never get it done. It happens sometimes. But you'll learn in the process.

Or just keep following the book and save that for after!

Summary

In this chapter, we've handled some loose ends, leaving us ready to set aside our first example project and move on to the next. We learned how to use the Build Settings window to build our project, and we learned about the Player Settings window, which contains properties relating to the appearance and behavior of our project once it is built.

Next, we'll move on to a project that will give you plenty of practice working with object-oriented programming concepts like inheritance.

PART II

Tower Defense

CHAPTER 26

Tower Defense Design and Outline

It's time to start our next project – a little tower defense game. We'll get some practice using inheritance, learn how to perform basic pathfinding for our enemy AI, get some experience with collision detection through scripts instead of colliders, and learn to use the newest UI features of Unity.

Gameplay Overview

If you're unfamiliar with the genre, a tower defense is a game where the player places structures (towers) on the playing field which defend against oncoming enemy attackers. One level at a time, enemies spawn in at a certain location and attempt to navigate their way to their goal. Whenever an enemy reaches the goal, you lose a life. This is sometimes referred to as “leaking.” Once you lose all of your lives, you lose the game. Using money gathered from winning levels and/or slaying enemies, you build more and more towers to prevent enemies from leaking and to stay in the game.

Our tower defense will have a simple setup: the playing field is a rectangle, longer than it is wide, with the player camera hovering over it and pointing down at it, a camera orientation much like in our first project.

Enemies spawn at the top of the playing field – the **spawn point** – and navigate to the bottom, the **leak point**. Your towers sit between them and their goal. They will navigate around your towers using basic Unity **pathfinding**, which is the act of finding a walkable route around all obstacles without touching or passing through them. Certain levels will instead have flying enemies which simply go over all our towers, no pathfinding involved.

Our towers can be positioned to route enemies along specific paths, particularly in the interest of keeping them in range of our strongest towers for a longer duration.

This is called “mazing” – the player will try to build an ideal maze that keeps the enemies in the right places for the longest time possible. Figure 26-1 shows the finished project with a small maze set up and some ground enemies navigating through it.

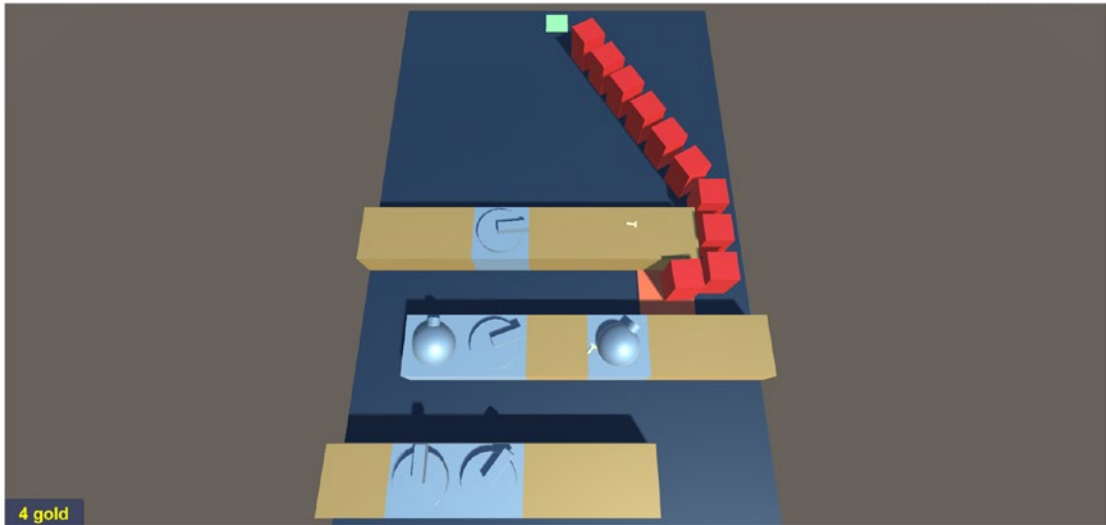


Figure 26-1. Ground enemies resembled by red cubes navigate through our maze while our arrow and cannon towers (blue) fire at them. The basic pale-brown cubes are barricades used to cheaply expand the maze

The game will transition back and forth between two states:

- **Build mode**, where there are no enemies spawning and the player can purchase and sell towers
- **Play mode**, where a round of enemies will spawn in sequence, one after the other, and attempt to reach the leak point.

The game will start in build mode so the player can construct their first towers. Once the player is ready to begin the level, they must press a button. At this point, we perform our pathfinding to check what route the enemies will take to reach the goal. If pathfinding cannot find a route through the towers, the player will be warned, and they won’t be able to start the level. Otherwise, if the player hasn’t blocked the path, we enter build mode.

Build mode will spawn the enemies, each one following closely behind the last. The level ends as soon as all the enemies have either been slain or have leaked. If the player’s health has dropped below 0 during this level, we gently break the news that they’ve lost. Otherwise, we enter build mode again, and the process can repeat itself.

Some tower defense games will have the enemies run through a path that the player can't build towers on – for example, the enemies might be down in a trench, while the towers can only be built on cliffs above. This solves the problem of the player being able to block enemies off with their towers and doesn't require any pathfinding. It's a game design thing, so I won't get into the differences between the two, but the concept of “mazing” isn't really a thing if you don't allow the enemies to mingle with the towers like we will. We want some practice with pathfinding anyway, so our method will suit us well.

The player earns money to build more towers as they slay enemies, as well as a chunk of extra money at the end of each round.

We'll implement a small variety of tower types:

- **Arrow Towers**, which rapidly fire a seeking projectile at a single targeted enemy at a time. Their projectiles are guaranteed to hit the target, homing in on them constantly. They can strike both ground enemies and flying enemies alike.
- **Cannon Towers**, which fire arcing projectiles at the ground where their target stands. Their projectiles deal damage in a radius, capable of hurting multiple enemies at once, but move slower and can be inaccurate. They do not fire at flying enemies. Since they fire at the ground location of their target, they might not even hit a target if it is quick enough.
- **Hot Plates**, which are flat towers that do not block enemies, allowing them to walk right over. They constantly apply damage to all enemies who stand on them. They're worthless against flying enemies.
- **Barricades**, which do not attack, but act as a cheap means of building a maze for enemies to navigate through. They're a price-efficient means of keeping enemies in range of your towers longer. Again, these are worthless against flying enemies.

Technical Overview

We're going to get into some new concepts in this project. It wouldn't be worth doing if it didn't challenge us in some new ways, right?

The movement will be handled first, some rudimentary stuff that isn't far off from what we've already done. The player is nothing but a floating camera in this project,

since they don't have one character they're controlling, but that camera must be able to move around the stage. We'll implement basic camera controls: arrow key movement, movement by dragging the mouse, and the option to scroll in and out with the mouse scroll wheel.

After that, we'll get our first taste of pathfinding. The process of pathfinding is a complicated topic, but Unity makes it a bit simpler for us. Luckily, that means we won't have to implement a pathfinding algorithm ourselves – although they can be a bit of fun if you're into that stuff. We'll learn how to work with this system to give our enemies a path around the towers.

We'll also learn the concept of raycasting. It's a means of detecting if a collision occurs along a given line, starting at one `Vector3` position and traveling along a given direction for a given amount of distance. This can be useful in a multitude of ways, but we'll be using it to detect the point on the playing stage beneath our mouse cursor so that we can point and click to where we want to place towers. In other words, we'll use it to get the world position on the stage beneath our mouse.

We'll also get some practice using inheritance. We learned about it before we started making our example projects, but this is the first time we'll put it to practical use. Using inheritance, we can reuse logic that certain towers share while still maintaining flexibility that allows us to define towers that behave differently than others (flame traps) or towers that don't do much of anything (barricades). Our projectiles will also have varying forms: some will seek a target and be guaranteed to hit it; others will arc and flop down on a targeted ground location. They're both still projectiles, sharing some logic, but they differentiate in specific ways that are implemented in their lower types. For our enemies, we'll have a base type that defines general stuff – notably, their health and the process of dying – and we'll implement lower types to contain the logic of ground enemies vs. flying enemies.

In build mode, we'll implement Unity's latest and greatest UI system to create something a bit more polished and complex than our GUI experience with our last project. We'll also get some experience with converting positions to points on a grid. All towers will have the same size, and we'll only allow the player to place them along increments of that size. You can place them perfectly side by side, with no space between them, or you can leave an entire tower's space between the two – but you can't do anything in between. This will require a bit of math-related tomfoolery, but it's not boring math. It's programming math – the fun sort.

At the end of every round, we'll give the player money for making it through, and we'll scale up the strength of our enemies. We'll have levels with flying enemies every fourth round, learning how to use a new operator to detect when that occurs.

Project Setup

To get started, create a new Unity project through the Unity Hub, using the 3D template and naming it TowerDefense, as shown in Figure 26-2.

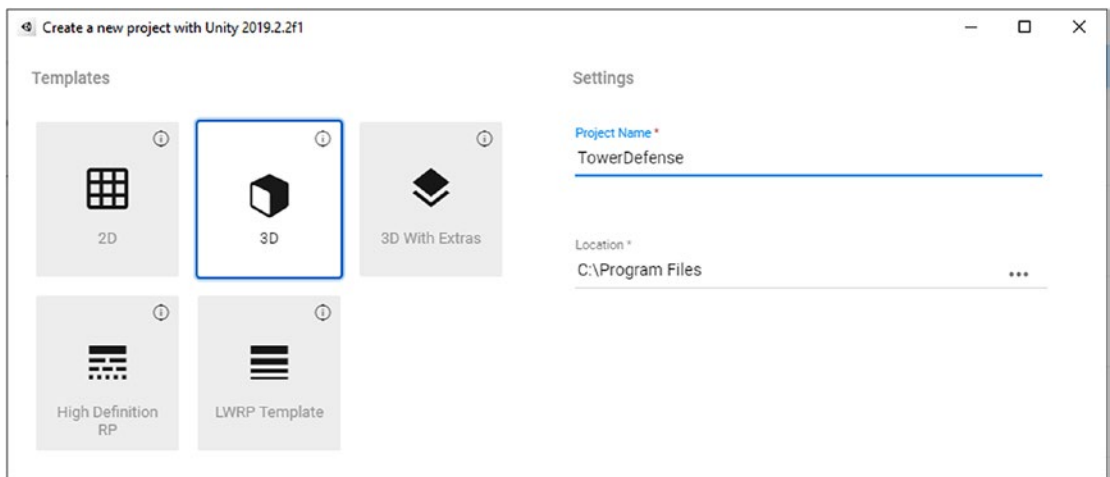


Figure 26-2. Creating our project through the Unity Hub

In the Project view, we'll create folders for Materials, Prefabs, and Scripts, all inside the Assets folder, which should already have a Scenes folder inside it. We'll rename the default "SampleScene" to "Main", just to sound more official.

Summary

We've gone over how our project is expected to play and briefly considered the details of implementation. Simply put, it's a basic form of the tower defense genre with enemies spawning on one side of the stage and attempting to reach the bottom, while the player-built towers attempt to stop them. We'll explore a lot of important concepts in the making of this project, like inheritance, raycasting, and pathfinding, so let's jump right in and start implementing the pieces one by one.

CHAPTER 27

Camera Movement

First things first, let's get some camera movement in working order. The player has to be able to move around the stage to be able to see the whole thing, after all. We'll allow the player to move the camera with the arrow keys as well as by right-clicking and dragging with the mouse. We'll also provide the option to use the scroll wheel to zoom in or out, which lets the player control how close their camera is to the stage.

All of the movement will be applied to a `Vector3` variable called `targetPosition`. Instead of directly applying the movement to the camera `Transform`, we update this vector variable with any changes in position we want to make to the camera. Every frame, we Lerp the camera toward that target position. This way, we can easily make the movement smooth and gradual with our Lerp call.

It also makes it easy to restrict the camera to a certain area. Before we move the camera toward the target position, we'll ensure that each axis of the target position is within a certain range. We define variables for the minimum and maximum X, Y, and Z values we want to allow the camera within. By setting these, we can prevent the player from accidentally moving the camera so far away that they can't see the stage anymore and get lost in the void. It will also prevent them from zooming too far in or out.

Setting Up

With your default scene open, let's set a few things up before we get rolling on the code.

Create a Plane. Name it Stage. This is the floor. Keep its position set to (0, 0, 0) and set its scale to (7, 1, 20). There. We've designed our level, and it only took us mere seconds. We sure are talented.

Now let's set up the camera. The camera that comes with the scene by default will work. We'll rename it to "Player Camera." We'll have it positioned over the level, pointing down at it with a slight upward angle so it's not pointing straight down. A Y position of 54 by default will work just fine. We'll use an X rotation of 70. A rotation of 90 will point straight down, so we just skew it 20 degrees off to give it that little tilt we want.

We'll make a Player script and attach it to the Player Camera GameObject. Let's declare some basic variables – a reference header, for now including only our quick Transform reference, and variables for the minimum and maximum X, Y, and Z values the camera will be confined to:

```
[Header("References")]
public Transform trans;

[Header("X Bounds")]
public float minimumX = -70;
public float maximumX = 70;

[Header("Y Bounds")]
public float minimumY = 18;
public float maximumY = 80;

[Header("Z Bounds")]
public float minimumZ = -130;
public float maximumZ = 70;
```

Those default settings will work for us, but we'll have them exposed in the Inspector in case we want to change them at some point. Save the changes to the script, go to the script component on the camera GameObject, and set that Transform reference to the Transform of the same GameObject. Remember, that's just a slightly faster means of referencing our own Transform, as opposed to using the ".transform" member that all scripts have. We'd more than likely be fine if we didn't do this, but it's a good habit to get used to.

We'll start by defining the methods we'll be calling to make things work. We'll use Start and Update, and we'll have some private methods that are called in Update. Just like with our Player in the first project, we'll be doing this simply to split the code up into neat blocks, keeping things clean and neatly separated.

After we declare the methods, we'll implement each one, one at a time. For now, though, we'll just leave them empty and make sure we're calling them in Update:

```
void ArrowKeyMovement()
{
}
```

```

void MouseDragMovement()
{
}

void Zooming()
{
}

void MoveTowardsTarget()
{
}

//Events:
void Start()
{
}

void Update()
{
    ArrowKeyMovement();

    MouseDragMovement();

    Zooming();

    MoveTowardsTarget();
}

```

This maps out our overall process. Just looking at the Update method, you can see how we plan on running things. With the methods named like they are, it practically spells itself out in plain English:

- First, check **arrow key movement**. If the player is pressing any arrow keys, the movement is applied to a Vector3 targetPosition variable.
- Then add **mouse drag movement** to the targetPosition as well, checking if the player is holding right-click and moving their mouse.

- Now add up and down movement caused by **zooming**. In world directions, this movement is up and down, so the Y axis, but if you're looking through the camera, it's like going forward or back, since the camera is pointing down at the stage.
- At last, we **move the camera toward the targetPosition**. We know the target position has been updated to match whatever input the player gave last, whether it be arrow key movement, mouse movement, or scrolling to zoom. Now we just restrict the target position to the bounds we declared as variables up at the top; then we Lerp the camera toward the target position.

Arrow Key Movement

To implement the arrow key movement, we'll write a simpler take on the code we wrote for the player movement of our first project. We use the `Input.GetKey` method to check if a key is pressed and apply movement directly to the `targetPosition` if it is.

First, we need to declare some variables. We still don't have a `targetPosition` variable, after all. While we're at it, we'll declare a variable for the sensitivity of mouse drag movement, which will come into play a bit later, as well as a variable for the amount of smoothing applied to the camera movement.

Add these variable declarations **under your existing variable declarations in the Player script**:

```
[Header("Movement")]
[Tooltip("Distance traveled per second with the arrow keys.")]
public float arrowKeySpeed = 80;

[Tooltip("Multiplier for mouse drag movement. A higher value will result
in the camera moving a greater distance when the mouse is moved.")]
public float mouseDragSensitivity = 2.8f;

[Tooltip("Amount of smoothing applied to camera movement. Should be a
value between 0 and 1.")]
[Range(0,.99f)]
public float movementSmoothing = .75f;
```

```
private Vector3 targetPosition;

[Header("Scrolling")]
[Tooltip("Amount of Y distance the camera moves per mouse scroll
increment.")]
public float scrollSensitivity = 1.6f;
```

The tooltips pretty much explain the variables. We're using a new attribute, `Range`, for our `movementSmoothing` variable. This attribute is used to make sure a number variable is clamped between certain values in the Inspector. It also adds a handy little slider control to the variable in the Inspector to make setting it easier. We use it to make sure the smoothing is clamped between 0 and .99f. The smoothing variable will be used when we Lerp the camera toward the target position. It's the third parameter in the Lerp call – the fraction. We'll use all that stuff later, though – and we'll talk about why we want the maximum value to be .99f instead of 1.

One thing that might slip the mind is that our target position is going to default to (0, 0, 0). This means the camera will always initially try to move there at the start of the game. To remedy that, all we need to do is set the target position to the camera position **in our Start method**:

```
void Start()
{
    targetPosition = trans.position;
}
```

This makes it so that whatever position we place the Player Camera at in the scene is the position it will start at.

Now, add this code **inside the ArrowKeyMovement method**:

```
//If up arrow is held,
if (Input.GetKey(KeyCode.UpArrow))
{
    //...add to target Z position:
    targetPosition.z += arrowKeySpeed * Time.deltaTime;
}
```

```
//Otherwise, if down arrow is held,
else if (Input.GetKey(KeyCode.DownArrow))
{
    //...subtract from target Z position:
    targetPosition.z -= arrowKeySpeed * Time.deltaTime;
}

//If right arrow is held,
if (Input.GetKey(KeyCode.RightArrow))
{
    //...add to target X position:
    targetPosition.x += arrowKeySpeed * Time.deltaTime;
}

//Otherwise, if left arrow is held,
else if (Input.GetKey(KeyCode.LeftArrow))
{
    //...subtract from target X position:
    targetPosition.x -= arrowKeySpeed * Time.deltaTime;
}
```

That'll do it. The `targetPosition` will have the arrow key movement applied to it, using `arrowKeySpeed` multiplied by `Time.deltaTime` to ensure it's "per second" instead of "per frame."

The movement won't actually apply yet, though. Let's remedy that so we can test our features as we implement them.

Applying Movement

To apply the movement, we'll simply clamp each axis of our target position (X, Y, and Z) between the minimum and maximum values and then Lerp the camera position toward the target position.

Rather than using if and else blocks and clamping the value between a minimum and maximum ourselves, which would take multiple lines of code for each axis, we

can use the method `Mathf.Clamp`. It takes three number parameters – the value, the minimum, and the maximum:

- If the value is below the minimum, the minimum is returned instead.
- If the value is above the maximum, the maximum is returned.
- Otherwise, if it's somewhere between, we just get the value returned back to us unchanged.

Add this code **in the `MoveTowardsTarget` method**:

```
//Clamp the target position to the bounds variables:
targetPosition.x = Mathf.Clamp(targetPosition.x,minimumX,maximumX);
targetPosition.y = Mathf.Clamp(targetPosition.y,minimumY,maximumY);
targetPosition.z = Mathf.Clamp(targetPosition.z,minimumZ,maximumZ);

//Move if we aren't already at the target position:
if (trans.position != targetPosition)
{
    trans.position = Vector3.Lerp(trans.position,targetPosition,
    1 - movementSmoothing);
}
```

We set each axis of our `targetPosition` vector individually, calling `Mathf.Clamp` for each one. This results in three pretty repetitive lines of code, the only difference in each one being the axis we refer to throughout: the letter X, Y, or Z. When you type this out, make sure you're getting it right – don't copy-paste and change only some of the Xs into Ys or Zs. That can cause some pretty confusing behavior!

Moving on, the movement is only applied if the camera (`trans.position`) is not (`!=`) already where it needs to be (`targetPosition`). When moving, we set the position to the result of our `Lerp` call. That call will move the camera toward the target position.

You'll notice the fraction we pass in as our third parameter in the `Lerp` call isn't just the `movementSmoothing` variable given as is. Rather, we give it "1 - `movementSmoothing`".

This is just so the variable makes more sense when you read its name and decide what value to give it. You would expect it to be smoother if the value is higher, right? A maximum `movementSmoothing` value should be the smoothest, while a minimum value should be the least smooth. If we gave the fraction as is, it would be quite the opposite.

Remember, this fraction given to the Lerp call is “How much of the distance between the two positions will we move?” If it’s a higher value, we move more. That means the camera will be snappy, getting where it’s going more quickly. A value of 1 will simply apply no smoothing at all. A low value like .05f will move much less per frame and, thus, the smoothing will be more noticeable.

By doing “1 – movementSmoothing”, we ensure that, if the smoothing is something like .8f, we get 1 – .8f. That’s .2f – a low value, resulting in more smoothing. In other words, we’re pretty much “flipping” the value around when we pass it to the Lerp call.

At this point, you can probably see why our Range attribute was set to keep the smoothing value at a maximum of .99f instead of 1. If the smoothing were allowed to be 1, then we would simply be giving “1 – 1” as the fraction, resulting in 0. This would mean the camera would never move at all! By giving it a maximum value of .99f, we ensure that we never accidentally break the movement that way. If you can predict some situation where your code keels over and breaks, you should probably not trust yourself to remember that it’s possible and avoid it in the future – because you probably won’t remember it. And then some months later, you’ll break it. And then you’ll spend two hours and a few pulled-out tufts of hair trying to figure out what went wrong. And kick yourself when you realize what caused the problem.

Anyway, you should now be able to test our arrow key movement in-game. Try playing with the smoothing value through the Inspector and note the difference when it’s lower or higher. A very high value makes the movement very slow and slippery, while a very low value makes it snappy at the risk of being somewhat unpleasantly jarring. I’ve set the default value to .75f because I feel it’s a comfortable middle ground – not so smooth that it’s clunky, but still smooth enough to notice it.

Mouse Dragging

Let’s get a move on. The next step is implementing camera movement.

Add this code **in the MouseDragMovement method**:

```
//If the right mouse button is held,
if (Input.GetMouseButton(1))
{
    //Get the movement amount this frame:
    Vector3 movement = new Vector3(-Input.GetAxis("Mouse X"),0,-Input.
    GetAxis("Mouse Y")) * mouseDragSensitivity;
```

```

//If there is any movement,
if (movement != Vector3.zero)
{
    //...apply it to the targetPosition:
    targetPosition += movement;
}
}

```

This first method call hasn't been used yet: `Input.GetMouseButton`. This is a simple one, much like `GetKey`. It checks if a specific mouse button is currently being held down. It uses an integer value as its parameter, identifying which mouse button to check:

- **0** is the **left mouse button**.
- **1** is the **right mouse button**.
- **2** is the **middle mouse button** (which can be pressed down for a click in most all mice).

This will return true while the right mouse button is held down and false while it is not.

We then declare a local `Vector3` named `movement`. We use another new `Input` method here: `GetAxis`. This is a more general means of reading input, where you provide a string as the means of identifying which kind of input you're after. In our case, we use "Mouse X" and "Mouse Y." These return what we call the "delta" for the mouse position, which means "how much it's moved from its last position." For example, if the user didn't move the mouse left or right on this frame, Mouse X returns 0. If they did, it returns some fraction representing how much they moved the mouse.

The values will work opposite to how we want them to affect our target position, though:

- **Mouse X** will be **positive** when the mouse cursor goes **right** and **negative** when it goes **left**.
- **Mouse Y** will be **positive** when the mouse cursor goes **up** and **negative** when it goes **down**.

If we apply it directly to the target position movement, we get an awkward result where the camera movement follows the direction of the mouse cursor. That's opposite to how you normally expect it to work when you drag your camera around in games like this. It sort of goes against the idea of "dragging."

That's why we have the “-” sign before each `Input.GetAxis` call. It flips the returned values around: if the delta was positive, it becomes negative, and vice versa. In other words, if the mouse cursor moves right, the camera moves left, and so on for the other directions. If you don't get this, take those “-” symbols out and try it yourself. You'll probably feel all sorts of wrong trying to move your camera with the mouse.

The `Vector3` we create is also correctly mapping the mouse axes to the actual 3D direction we want to move our camera. Remember, we're operating on world direction here, not a direction local to the camera. The Mouse Y isn't actually “up and down” in-game, since the camera is pointing down. To go “up” from the perspective of the camera, we really need to go forward in world directions. Thus, the mouse Y movement needs to correspond to the Z axis when we move the camera, so we provide it as the Z axis when we create the vector, and we leave the Y axis at 0 so that no movement occurs there.

After the vector is created, we multiply it by our `mouseDragSensitivity` variable, giving us an easy way to adjust the amount of movement the mouse generates for the camera. Since it's tied to a variable like this, we could later implement a way for the player to set this themselves, such as through an options menu in-game.

After this, all that's left to do is apply the movement to `targetPosition`, assuming there was any movement.

That's all we need to make this feature work. You can test it out if you like. Note how changing the sensitivity variable affects the movement.

Zooming

To implement the zooming, we need to detect mouse scrolling. We can do this with a new member of `Input` by the name of `mouseScrollDelta`. This is a `Vector2`, which is just like a `Vector3` except that it only has X and Y axes – no Z. The X axis corresponds to the mouse wheel being pressed left or right, which some mice support (but not all of them). It'll be -1 if the wheel was ticked left or 1 if the wheel was ticked right. But we don't need that function. The Y axis is what we're after: it measures how much the wheel has been scrolled up (positive) or down (negative) this frame.

Like with the mouse dragging, that value doesn't correspond to the movement we want. We want to go down toward the stage whenever the mouse wheel is scrolled up, which means we need to decrease our Y position. We want to go up when the wheel is scrolled down, increasing our Y position. The way it is by default, we'll do the opposite. Again, we can just fix this by inverting, or “flipping,” the Y value with a “-” sign.

You'll see it in the following code, which you'll be writing **in the Zooming method**:

```
//Get the scroll delta Y value and flip it:
float scrollDelta = -Input.mouseScrollDelta.y;

//If there was any delta,
if (scrollDelta != 0)
{
    //...apply it to the Y position:
    targetPosition.y += scrollDelta * scrollSensitivity;
}
```

As is usually the case with operator stuff, we could accomplish the same “flipping” effect if we subtracted from the Y axis, changing that “+=” into a “-=”. It doesn’t really make a difference how we do it, just so long as we make sure it gets done!

With that in place, we now have all of our movement options up and running: arrow keys, right-click mouse dragging, and scroll wheel to zoom in and out. It’s all clamped with our bounds variables, keeping the player within view of the stage at all times, and we use some configurable smoothing to add that feeling of luxury.

Summary

This chapter got our player camera movement working, exercising some concepts we’ve worked with already and teaching us a few new tricks, primarily relating to detecting mouse input. Here’s a rundown of things to remember:

- The **Mathf.Clamp** method takes three number parameters (int or float): a value to clamp, a minimum, and a maximum. It will return the value, but will clamp it to never be lower than the minimum or greater than the maximum.
- The **Input.GetMouseButton** method returns true if a given mouse button is being held on this frame. It takes an int value to identify which mouse button is in question: 0 is the left mouse button, 1 is the right mouse button, and 2 is the middle mouse button.

- **Input.GetAxis(“Mouse X”)** returns the mouse X movement on this frame: **positive** when the cursor moves **right** and **negative** when it moves **left**.
- **Input.GetAxis(“Mouse Y”)** returns the mouse Y movement on this frame: **positive** when the cursor moves **up** and **negative** when it moves **down**.
- The **Input.mouseScrollDelta** property returns a Vector2 (a vector with just X and Y values) representing mouse scrolling that has occurred on this frame. The **X axis** is **left and right** movement, which not all mice support, and the **Y axis** is **forward and back** scrolling – the standard scrolling that all mice should support.

With that out of the way, we can focus on the core mechanics of towers, projectiles, and enemies in our next chapter.

CHAPTER 28

Enemies, Towers, and Projectiles

In this chapter, we'll create our first tower, the arrow tower, and set up the groundwork for towers firing projectiles at enemies. To accommodate this, we'll also set up the base for our enemies, giving them health points and making them “die” when their health runs out. They won't move yet, but they'll give our towers something to shoot at.

These features will give us plenty of practice with the concept of inheritance. We'll learn how to share the common functionality of an upper type, like a generic enemy, with the more specific implementation of lower types, like ground enemies and flying enemies.

Layers and Physics

Before we begin, let's set up the layers we'll be working with so they'll be ready to use later. Navigate to Edit ► Project Settings ► Tags and Layers. Under the Layers dropdown, add these four layers to the given “user layer” fields:

- **User Layer 8:** Enemy
- **User Layer 9:** Tower
- **User Layer 10:** Projectile
- **User Layer 11:** Targeter

By the end, it should look like Figure 28-1.

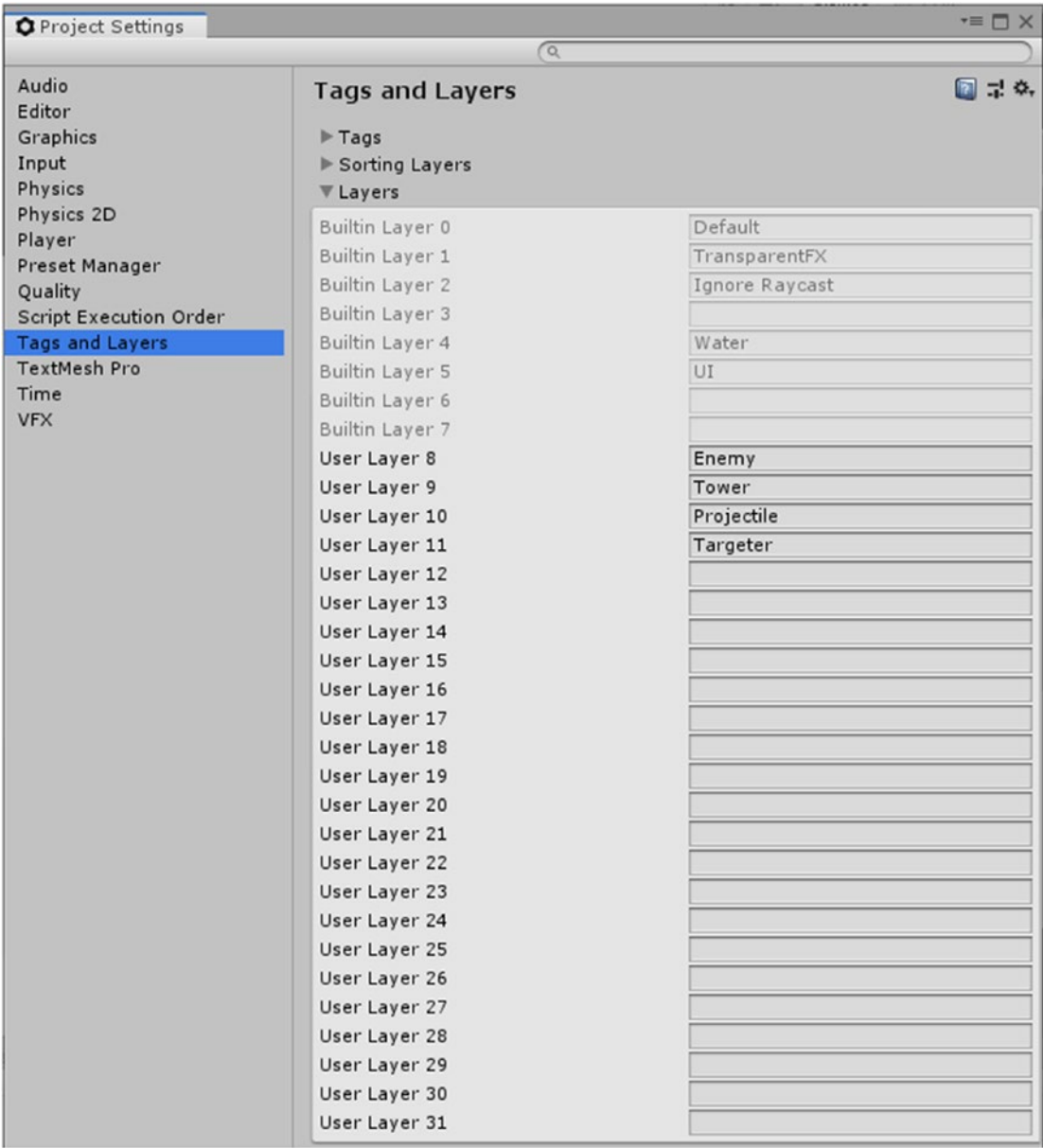


Figure 28-1. Our layer settings in the Project Settings window

While we have the Project Settings open, let’s also set up the physics collision detection matrix. We did this for our last project too – it’s how we make sure certain layers only collide with layers that relate to them. In that same Project Settings window, click the Physics tab on the left, sitting above the Tags and Layers tab we just used.

We'll set ours up as shown in Figure 28-2. Targeters will be used to detect enemies that come within range of towers, so we want them to only collide with the Enemy layer.

Aside from that, nothing else is that important. We'll make our Projectile layer only collide with Enemies and make our Enemies not collide with each other or with Towers. It isn't a big deal, anyway – we don't really plan on using collisions for this project. Our projectiles won't actually strike enemies, as far as the collision system is concerned, because we're just scripting them to reach a certain point and then disappear. Our enemies won't be stopped by collisions, since we're moving their Transforms directly along a predetermined path. But we'll set it up this way to be prim and proper.

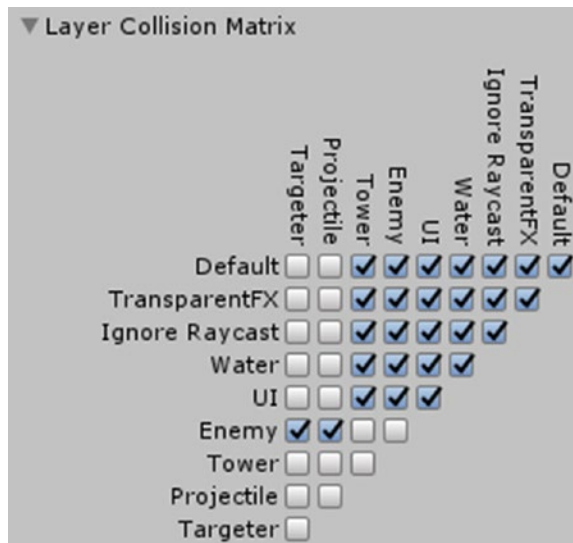


Figure 28-2. Our settings for the Layer Collision Matrix field, found in the Physics tab of the Edit ► Project Settings window

Basic Enemies

For now, we'll be creating a **base class** for our enemies. This will hold logic that both ground enemies and flying enemies will use, but not the specific stuff that differentiates between them – notably, how they move. Every enemy needs to have health points and a means of taking damage. They'll also need to die when their health runs out. Since both types of enemies share this logic, we'll put that in the base class and implement their movement in separate subclasses. These subclasses are `FlyingEnemy` and `GroundEnemy`. We'll code them later.

Create an Enemy script and let's declare some variables:

```
[Header("References")]
public Transform trans;
public Transform projectileSeekPoint;

[Header("Stats")]
public float maxHealth;
[HideInInspector] public float health;

[HideInInspector] public bool alive = true;
```

The first four variables are simple enough:

- **trans** – A quick reference to the Transform of the root GameObject.
- **projectileSeekPoint** – A reference to a Transform which will resemble where seeking projectiles should home in on. The Enemy script will be on the root GameObject, which will be positioned on the floor level. We don't want homing projectiles to aim for our enemies' feet, though, so we want to expose a separate Transform that our projectiles will aim for.
- **maxHealth** – Maximum amount of health an enemy can have.
- **health** – The current health the enemy has. We want our enemies to always spawn at maximum life, so we don't let ourselves set this variable in the Inspector, giving it the HideInInspector attribute. However, we keep it public so other classes can see how much health the Enemy has if they ever need to.

Finally, we have the **bool alive** variable. When an enemy dies, we'll be calling the Destroy method to remove the enemy GameObject from the world. Sometimes, this doesn't necessarily happen instantly. It might take until the end of the frame (after all of our Update logic goes through) for the enemy, and thus its Enemy script, to be destroyed.

This means we might have a case where an enemy is technically dead, but not quite gone yet. Thus, we keep track of their state in this variable so we can always make sure they're actually alive before we do anything to them with other scripts, like towers.

We'll need the current health to be set to the maximum health in a Start method; otherwise, it'll stay at 0 all the time, so declare this method in your Enemy script:

```
protected virtual void Start()
{
    health = maxHealth;
}
```

We've heard of the **protected** keyword before. It's used in place of public or private, and means that this member can only be accessed by this class or any class which inherits from this class. If we made it private, we'd be unable to mess with this method at all in the inheriting types we plan on declaring later, GroundEnemy and FlyingEnemy.

But what is **virtual**?

The virtual keyword makes our method overridable by lower types. This means that a class which inherits from Enemy, such as our GroundEnemy or FlyingEnemy, can declare their own Start method that "overrides" this one. Overriding the method means declaring their own logic that runs when the Start method is called, so that Enemy.Start runs first and then their code runs after. This is only possible if the method is marked as virtual – and the inheriting type has to have access to the method, which is why we made it protected instead of private. It could also be public.

We'll see an example of how to override a method in a bit, but the point here is that we've made the method virtual to ensure that inheriting classes can still declare their own logic that happens on Start. If we didn't make this method virtual, then declaring another method named Start in a script that inherits from this one would "hide" the original declaration. Only one of them would actually be called by Unity when the script is initialized – the lower type, the one that inherits from Enemy, would get called. Thus, we must declare the method as virtual to ensure that if one of our lower types wants to run their own Start logic, they can do so by overriding the method, rather than "hiding" it and causing the original Start not to run.

Now let's declare a method in the Enemy script to make the enemy take damage. It's a simple method that takes life away from the enemy and then checks if the enemy has just lost the last of their remaining life. If so, it calls a second method, Die, which destroys the enemy and sets "alive" to false:

```
public void TakeDamage(float amount)
{
    //Only proceed if damage taken is more than 0:
```



```

    if (amount > 0)
    {
        //Reduce health by 'amount' but don't go under 0:
        health = Mathf.Max(health - amount,0);

        //If all health is lost,
        if (health == 0)
        {
            //...call Die:
            Die();
        }
    }
}

public void Die()
{
    if (alive)
    {
        alive = false;
        Destroy(gameObject);
    }
}

```

We don't want to "kill the enemy twice," so we make sure they're alive before we proceed to Destroy the GameObject that the Enemy script is attached to – which will be the root GameObject of the enemy. Of course, we also mark their death by setting "alive" to false.

All that's left now is to create a temporary enemy to test with. We'll replace it with a more specific version down the road, but for now, we just want something we can test the Enemy script out on:

- Create an empty GameObject named Test Enemy. Position it anywhere on the floor, keeping its Y value at 0. Set its layer to Enemy.
- Add a Rigidbody, mark it as kinematic, and add a non-trigger box collider. Set the collider size to (5, 8, 5) and set its center to (0, 4, 0).

- Right-click the Test Enemy in the Hierarchy and add a Cube child. Remove its BoxCollider component; we want the collider to be on the root only. Set its scale to the same size we gave that collider: (5, 8, 5). Set the local Y position to 4. It should now be covered by the collider. Its layer should also be Enemy.
- Add the Enemy script to the root GameObject (named Test Enemy). Assign the “trans” reference to the root Transform, and assign the “projectileSeekPoint” reference to the Cube Transform. This will make projectiles home in on the center of the cube; if we had made it the root Transform, they’d home in on the very bottom of the cube. Set the Max Health to 12.
- I’ll make an Enemy material and apply it to the Cube, giving it a red color with a hex value of DD1717.

Projectiles

To implement projectiles, we’ll be using inheritance again to declare a base Projectile class and then two separate subtypes: SeekingProjectile and ArcingProjectile. Every projectile deals a certain amount of damage, travels at a certain speed, and targets a certain Enemy that’s being fired at. The difference is how they move and how they inflict that damage.

Seeking projectiles home in on the target enemy’s projectileSeekPoint Transform, which we made a public reference to earlier for just this purpose. They move at a set speed, directly toward that point, until they reach it. Once they reach it, they damage the enemy and destroy themselves.

Arcing projectiles lock in their target location as soon as they are spawned. They aim for the base Enemy Transform, which is on the floor level, because we want them to look as though they are being lobbed and landing on the floor, where they “blow up” (without any cool special effects, unfortunately) and deal damage to all enemies around that impact point. They won’t be targeting flying enemies, so we don’t have to worry about how they react to enemies that aren’t positioned on the floor.

Create a Projectile script. The first thing we’ll do is add the **abstract** keyword to the class declaration line:

```
public abstract class Projectile : MonoBehaviour
```

This is an inheritance-related keyword. When we mark a class as **abstract**, we are implying that the class itself is not meant to have instances created and directly worked with. Rather, it acts as an upper type for more specific classes that inherit from it – but the class itself is never instanced directly. It merely serves as a base for other classes which share similar purposes.

Once you add that “abstract” keyword to make the class declaration line look like the preceding code, save the script and go to the Unity editor. Try to drag and drop the Projectile script onto a GameObject in your scene. Once you drop it, Unity will give you a popup error message:

Can't add script behaviour Projectile. The script class can't be abstract!

The class is abstract and thus should not be created directly. The only way to create an instance of a script class is by adding it as a component to a GameObject – but it's abstract, so Unity will prevent us from doing that. However, once we create our SeekingProjectile and make it inherit from Projectile, we will be able to add it, because we won't mark that one as abstract.

As such, declaring a class as abstract not only makes it clear that it is not meant to be used as is but also enforces that by preventing us (or others using our code) from misusing it at some point down the road.

Now let's write the contents of our base **Projectile class**:

```
[HideInInspector] public float damage;
[HideInInspector] public float speed;
[HideInInspector] public Enemy targetEnemy;

public void Setup(float damage,float speed,Enemy targetEnemy)
{
    this.damage = damage;
    this.speed = speed;
    this.targetEnemy = targetEnemy;

    OnSetup();
}

protected abstract void OnSetup();
```

First, you'll notice we have HideInInspector attributes given to our three variables. You might think that we would have the damage and speed variables show in the Inspector,

but that's not how we'll be setting the values up. We don't want to tie the damage to our projectile script instances: we want the tower that spawns the projectile to set the data.

Next, you'll notice our intended method of applying the damage and speed: through this Setup method, which simply takes "damage" and "speed" parameters and applies them directly to the projectile's variables.

This will be called by the tower instance that fires the projectile. We want the towers to determine the damage and speed themselves. That way, we can have variations of towers which deal more damage or fire quicker projectiles, without having to create a separate projectile prefab with those settings on each one. For example, we might have a bunch of variations of arrow towers, where each one gets stronger, gaining more damage and firing quicker projectiles – but they all use the same projectile prefab, because they all fire the same arrow. If we set the damage on the Projectile script itself, we'd have to make a separate prefab for each type of arrow tower.

Similarly, if we ever get to a point where we want to change a tower's stats on the fly in-game, we can do so: since the tower passes the damage and speed to each projectile it creates, as soon as we change the tower's damage and speed, any new projectiles it creates will reflect that change. This would be useful if you wanted to, say, implement a tower that strengthens all towers within a certain range of it by boosting their damage.

And finally, after setting our variables in the Setup method, we call another method: OnSetup. This is a method we declare in the following, but it's different than any other methods we've declared so far. It's marked with the abstract keyword, just like our class, and it has no code block coming after it. It has no curly braces – just a method declaration with no parameters and then a semicolon.

An abstract method is much like a virtual method. It can be overridden by lower types, but unlike virtual methods, an abstract method cannot provide its own implementation – which is why it has no code block after it. As well as this, an abstract method can only go in an abstract class, and any inheriting classes **must** implement an abstract method. Even if the inheriting class doesn't actually want to run any code when the method is called, it still must declare the method.

So this OnSetup method we declare is a means of giving our lower types a method that they can override to declare their own logic for what happens after the projectile sets up. By declaring the method, we give ourselves something that can be called, but we don't specify any code to run when it's called. That's for our lower types to decide. It's something like an event, akin to the Start and Update methods Unity provides us, except this time we've declared it ourselves!

Our SeekingProjectile doesn't actually need this logic, but our ArcingProjectile will. It needs to be told when its targetEnemy has been set so that it can mark the position of the enemy as soon as possible. That's how it knows where it will land. If we just did that in a Start method, the targetEnemy would still be null because the Setup method doesn't get called until after the Projectile is created by the tower!

We'll see how to override the method later.

We also mark this method as protected, giving our lower types access to it, but not public because other classes shouldn't be able to call it – they have no reason to.

Now let's implement the projectiles our arrow towers will use. Create a new script named SeekingProjectile. Change the declaration line to this:

```
public class SeekingProjectile : Projectile
```

The only difference from the default is that we've made it inherit from our "Projectile" script instead of "MonoBehaviour", the base class for all scripts. Now our script has the same members as Projectile: damage, speed, and targetEnemy.

This will generate an error because we have inherited from Projectile, but we have not implemented the abstract OnSetup method that Projectile expects us to. We'll handle that in a bit.

The behavior for our seeking projectiles is somewhat simple. They're fired at a specific Enemy and will travel toward them until they touch the enemy "projectileSeekPoint" transform. If the enemy dies while the projectile is traveling, it will continue to the point the enemy was at when it died.

To accomplish this, we'll store the enemy projectileSeekPoint position every Update call to keep track of where the Enemy was last positioned, as long as they're still alive. We'll move toward this position. Even if the enemy has died, we'll always have its last position stored.

We'll declare these variables:

```
[Header("References")]
public Transform trans;

//Private variables:
private Vector3 targetPosition;
```

Now let's set that `targetPosition` variable on every frame and move the enemy. We'll put this **in the Update method**:

```
if (targetEnemy != null)
{
    //Mark the enemy's last position:
    targetPosition = targetEnemy.projectileSeekPoint.position;
}

//Point towards the target position:
trans.forward = (targetPosition - trans.position).normalized;

//Move towards the target position:
trans.position = Vector3.MoveTowards(trans.position, targetPosition, speed *
Time.deltaTime);

//If we have reached the target position,
if (trans.position == targetPosition)
{
    //Damage the enemy if it's still around:
    if (targetEnemy != null)
        targetEnemy.TakeDamage(damage);

    //Destroy the projectile:
    Destroy(gameObject);
}
```

This is all the logic we need for the `SeekingProjectile`, but if we save and check the Unity Console, we'll see an error message:

CS0534: 'SeekingProjectile' does not implement inherited abstract member 'Projectile.OnSetup()'

As we discussed before, this is because when an upper class, like our `Projectile` script, has an abstract method declaration in it, all lower types, like `SeekingProjectile`, must provide an implementation of that method. They have to override it. Even if they don't want to actually run any code in that method, they still must have a declaration for it.

The declaration will be quite simple:

```
protected override void OnSetup(){}
```

Just throw that in our `SeekingProjectile` script class. We declare the same method, `OnSetup`, with the same protected access modifier, but this time we specify it as an “override.” This means we’re declaring it as an override to a method by the same name in the upper type. We can write whatever code we want in our version of the method, and whenever `OnSetup` is called from `Projectile.Setup`, our code will run. But we don’t want to actually do anything in the method call, so we simply put an empty code block `{}` at the end. Since it has a code block, we don’t need a semicolon at the end.

That should do it. We can’t test our projectiles until we have towers to fire them, but let’s set up an `Arrow` prefab so it’s ready to use in our arrow tower.

We’ll create a very blunt-looking arrow out of two cubes:

- Create an empty `GameObject`. Name it `Arrow` and set its layer to `Projectile`.
- Attach a `SeekingProjectile` script instance to the `Arrow`. Set the “trans” reference to point at the `Transform` of the `Arrow`.
- Right-click the `Arrow` and create a child `Cube GameObject`. Name it `Shaft`. Set its local position to `(0, 0, .75)` and set its scale to `(.4, .4, 1.5)`.
- Create another `Cube`, this time a child of the `Shaft`. Name it `Head`. Set its local position to `(0, 0, .625)`. Set its local scale to `(3, 1, .25)`.
- I’ll create an `Arrow` material and assign it to both cubes, giving it a yellow color with a hex value of `F2F89F`.

When you’re finished, it should look something like [Figure 28-3](#).

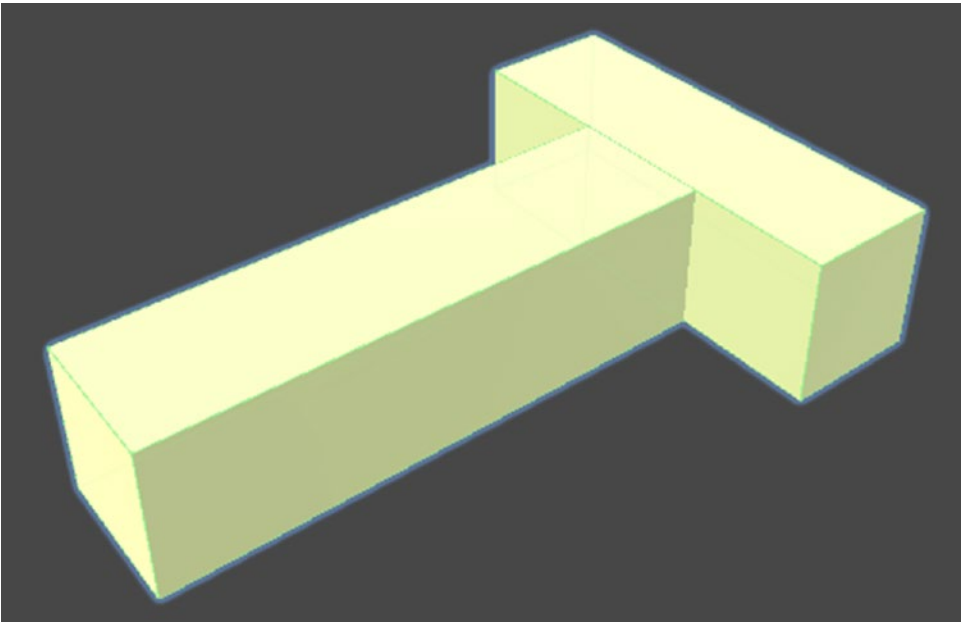


Figure 28-3. *Our Arrow GameObject*

With that, we can create a prefab for the Arrow and remove it from the scene.

Targeters

We'll use a system where our towers have a "targeter" to detect enemies within range. Using trigger collisions, targeters will detect when enemies touch their collider, which can be a sphere collider or a box collider, and will store those enemies in a List. When enemies leave the collider, it removes them from the List.

Each tower will have a Targeter GameObject nested inside it, which is an empty GameObject with the collider attached, as well as the Targeter script we're about to write.

Towers will use a reference to their Targeter to figure out which enemy they want to attack next. Arrow towers and cannon towers will have targeters that cover their attack range and will use the list of enemies as a collection of valid targets. In the case of the "hot plates" we'll be coding later, they'll use a box collider covering just the area taken up by the plate itself and constantly drain health from enemies detected by the targeter.

To detect when enemies touch the Targeter, we'll use the OnTriggerEnter event, which we used in the first project as well (to code the Hazard script).

Create a new script named `Targeter`. We'll only need two variables:

```
[Tooltip("The Collider component of the Targeter. Can be a box or sphere
collider.")]
public Collider col;

//List of all enemies within the targeter:
[HideInInspector] public List<Enemy> enemies = new List<Enemy>();
```

First, we have a reference to the Collider of the `Targeter`. The Collider type is the base class for Unity's built-in collider components. In our case, we expect it to be a `SphereCollider` or a `BoxCollider`. If you'll recall from Chapter 11, these components are both lower types of the Collider class (they inherit from it), so the Collider-type field can store a reference to them. That's the magic of inheritance, and we'll be seeing more of it later. You can always store an object, like a Box or Sphere Collider, as a reference in an upper, "less specific" type, like a Collider.

A Box Collider is more specific than a generic Collider. It uses the members of a Collider, but also adds its own members and does extra stuff itself. However, we can still store it as a Collider, because we know that it has all the same members as a Collider. Nothing is going to go wrong if we access those members. It's guaranteed to have them. But if we want to access the Box Collider members, we have to "cast" it back to a Box Collider, because once we store it as a Collider, the compiler is no longer sure exactly what type it is. You'll see an example of how all this works in a second, when we actually interact with the Collider.

Second, we also declare a List we'll be using to store all enemies that are currently within that collider. If you'll recall, Lists are like arrays, but we can add and remove items from a List on the spot, and the size of the List automatically updates whenever we do. When an enemy touches the collider, we add them to this List. When an enemy leaves the collider, we remove them from this List. We initialize the List as a new instance when we declare it. Since it's a List, not an array, we don't have to specify the size. It can store however many enemies we need it to store.

We also make it hidden in the Inspector. Lists can be serialized and thus shown in the Inspector to allow us to set them up with an initial collection of items. While this can be useful, we don't want or need it in this case. We want the List to be handled in-game by our code only.

We'll also declare a handy property that returns true if there are enemies within the Targeter or false if there are not:

```
//Return true if there are any targets:
public bool TargetsAreAvailable
{
    get
    {
        return enemies.Count > 0;
    }
}
```

Your first idea of how to implement this might be to do an “if” to check if the Count is greater than 0 and, if so, “return true;” or else “return false;”

But the way we've done it works too – and in just one line of code. Remember, the > operator simply takes a number on each side and returns true if the left number is greater than the right. It's already returning a bool – the data type our property should return. We can return the result of that operator as is, no if's or else's required.

This property may seem redundant, but it's a nice way to make your code read like plain English. Whenever a tower needs to ask its targeter if there are any targets available, rather than typing “if (targeter.enemies.Count > 0)”, we can just type “if (targeter.TargetsAreAvailable)”. It makes it obvious what we're asking – granted, it was already somewhat obvious, but more clarity never hurt anyone!

Moving on, we want to automate the way towers set their Targeter size. Towers which fire projectiles will have a “range” stat of their own. Rather than fiddling with the size of their Targeter colliders to depict their range, we'll just set that “range” variable and have our scripts automatically size the colliders based on it. For that, we need a Targeter method that can set the size based on range.

Box Colliders and Sphere Colliders have different means of measuring their size. Box colliders have a “size” Vector3 depicting the width, height, and length of the box. Sphere colliders instead use a “radius” float depicting the radius of the sphere – in other words, the distance from the center of the sphere to its edge.

The Collider type itself doesn't deal with the size at all, and that's the type we store our reference “col” as. As far as the compiler is concerned, a Collider doesn't have those variables. A BoxCollider has a size. A SphereCollider has a radius. But a simple Collider has neither. So we'll have to cast the Collider reference to a Box Collider or Sphere Collider – whichever it actually is – to be able to access the size/radius member.

Let's observe one way we might implement this, using "is" to check the type and then "as" to cast the Collider to a more specific type. Write this method in the Targeter script:

```
public void SetRange(int range)
{
    if (col is BoxCollider)
    {
        //We multiply range by 2 to make sure the targeter covers a space
        'range' units in any direction.
        (col as BoxCollider).size = new Vector3(range * 2,30,range * 2);

        //Shift the Y position of the center up by half the height:
        (col as BoxCollider).center = new Vector3(0,15,0);
    }
    else if (col is SphereCollider)
    {
        //Sphere collider radius is the distance from the center to the
        edge.
        (col as SphereCollider).radius = range;
    }
}
```

We use "is" to return true if "col" is pointing at a BoxCollider or false if it is not. If so, we cast it to a BoxCollider, in a set of parentheses (), using the "as" keyword. This is safe to do now, because we know that it is a BoxCollider – we just checked. We can then reach into it to access the "size" member. We're telling the compiler to look at it as a BoxCollider, not a Collider.

The same process goes for the sphere collider. If the collider isn't a BoxCollider, we'll instead check if it's a SphereCollider. If so, do the cast, reach in, and access "radius."

For box colliders, we always give them a height value of 30 units, which should cover enough space above the targeter to reach any flying enemies we have later on. The center has to be adjusted to shift the collider upward by half the height, so its bottom lines up with the Targeter instead of its center – otherwise, half of it will stick through the floor. This way, we can leave the targeters at a Y position of 0, aligned with the floor, and their colliders will size and position themselves appropriately.

This way of doing it is technically a bit slower than it could be. We’re casting to a `BoxCollider` three times: once to check if it “is” a box collider, then again to set the size, and then again to set the center. We then cast to a sphere collider to check if it “is” one, and if so, we cast it again to assign the value.

For our purposes, this method works fine. It does what we need it to do. It’s not likely to cause a noticeable performance hit. However, there are some situations where being conscientious about these things can make a difference – for example, in a loop that’s occurring thousands of times. Casting from one type to another is not free. It spends a little bit of our processing power.

If we wanted to be extra vigilant, we could perform the cast once with the “as” operator, storing the result in a local variable:

```
BoxCollider boxCol = col as BoxCollider;
```

This will result in “null” if the Collider is not actually a `BoxCollider`. We can use that in place of the “is” we used before, simply checking if the `boxCol` is null. If it is null, then it’s not a box collider. If it is not null, we can proceed – and we’ve already cast it to the local variable, so we can avoid casting it twice more when assigning the values by using “`boxCol`” instead of “`(col as BoxCollider)`”. After that, we could just run through the same process to check if the collider is a `SphereCollider` instead.

All in all, it would use less casts to get the same result – but the code would be a little longer and a bit clunkier.

With that out of the way, we now have a means of setting up the `Targeter` range (and, in doing so, the tower range) with a method, and it’ll automatically work regardless of the collider type the `Targeter` is set up with. We’ll call it once we get around to implementing the towers themselves.

Next, we need to keep track of enemies, properly adding and removing them from our `List<Enemy>`.

We’ll write the `OnTriggerEnter` built-in Unity event in our `Targeter` script:

```
void OnTriggerEnter(Collider other)
{
    var enemy = other.gameObject.GetComponent<Enemy>();

    if (enemy != null)
        enemies.Add(enemy);
}
```

When a collider enters ours, we declare the “enemy” local variable which attempts to grab an Enemy component from the same GameObject that the collider was attached to. We use the shorthand “var” for the variable type, just to be lazy. The compiler knows that we’re expecting to get an Enemy returned to us, so it figures out what type we want the variable to be.

The GetComponent method will return null if the component was not found. We check if we successfully found an Enemy component and, if so, add it to the “enemies” List.

You’ll recall that, when we set up our test Enemy GameObject, we made sure that the Collider was part of the root GameObject, not attached to the Cube within. This was to ensure that it’s easy for us to grab the Enemy component. Since they’re both on the same GameObject, we can reliably call GetComponent on the same GameObject that the collider is attached to. If we had the collider attached to the Cube, then it wouldn’t detect the Enemy component at all, because that’s on the root GameObject.

A very similar process will be used to remove enemies from the List when they exit the collider. We write an OnTriggerExit method and use the same means to look for the Enemy component, but this time, call the Remove method of our List instead of the Add method:

```
void OnTriggerExit(Collider other)
{
    var enemy = other.gameObject.GetComponent<Enemy>();

    if (enemy != null)
        enemies.Remove(enemy);
}
```

The Remove method will remove the given instance from the List if it actually exists in the List. If not, it simply does nothing.

With the targeted enemies being tracked, we can now implement a method that our towers can use to find the enemy that’s closest to them. We can do this by looping through the enemies in the List, checking the distance between their position and the tower position, and keeping track of which Enemy had the lowest distance. We’ll use a local variable for the lowest distance we found so far and another local variable to store the Enemy who had the lowest distance.

But we have another problem to solve. If an enemy dies while inside the List, the List won’t automatically remove them. The Enemy will become “null” in the List, but it will

still be an item in the List, taking up an index. Thus, we must be ready to deal with null references when we loop through the items in our List. We'll have to remove the null ones from the List as we go.

This brings forth another problem. When you remove from a List while looping through the List, you must be conscious about the effect that will have on the indexes stored in the List. Every item in the List that's stored "ahead" of the removed item will be shifted back to account for the removed item. Their indexes all decrease by 1 point.

We'll go over an example of this. Here's a basic loop that goes through the List of enemies, storing the current Enemy in a local variable and either removing them from the List if they're "null" or doing some code on them if they're not null:

```
//Loop through enemies:
for (int i = 0; i < enemies.Count; i++)
{
    var enemy = enemies[i]; //Current enemy

    //If the enemy has been destroyed:
    if (enemy == null)
    {
        //Remove it from the list:
        enemies.RemoveAt(i);
    }
    else //If the enemy is still around
    {
        // [do something with the enemy]
    }
}
```

The List.RemoveAt method is like Remove, but instead of taking an Enemy instance to remove, it just takes the index of the item we want to remove.

This might look like a fine way of doing it, but it messes with our indexes and causes some unwanted behavior. Let's say, for example, our List has indexes 0–5, totaling six enemies.

Now imagine indexes 0, 1, and 2 go by with no problem, but then, when our "i" is at 3, we find an enemy that's null. We now have indexes 4 and 5 ahead of us, still left to operate on.

So we remove that enemy at index 3. Now the enemies ahead of it are shifted back by the List. Index 4 becomes index 3. Index 5 becomes index 4.

Our “i” is still set to 3. Our loop iteration finishes, and the “for” increments “i” by 1 again. We’re now at an “i” value of 4.

We’ve completely skipped an item! The item that was at index 4 was shifted to 3, but we skipped past index 3 and moved right along.

The solution is simple enough. We just subtract 1 from “i” after we remove the item from the List. Then, the “for” loop will add 1 when the iteration finishes, and we end up back at the same index instead of skipping it.

With that complication settled, let’s write our method. Add this to the Targeter script:

```
public Enemy GetClosestEnemy(Vector3 point)
{
    //Lowest distance we've found so far:
    float lowestDistance = Mathf.Infinity;

    //Enemy that had the lowest distance found so far:
    Enemy enemyWithLowestDistance = null;

    //Loop through enemies:
    for (int i = 0; i < enemies.Count; i++)
    {
        var enemy = enemies[i]; //Quick reference to current enemy

        //If the enemy has been destroyed or is already dead
        if (enemy == null || !enemy.alive)
        {
            //Remove it and continue the loop at the same index:
            enemies.RemoveAt(i);
            i -= 1;
        }
        else
        {
            //Get distance from the enemy to the given point:
            float dist = Vector3.Distance(point, enemy.trans.position);
```

```

        if (dist < lowestDistance)
        {
            lowestDistance = dist;
            enemyWithLowestDistance = enemy;
        }
    }
}

return enemyWithLowestDistance;
}

```

The method will return an `Enemy`, or null if no enemies are in the `Targeter`, and take a single `Vector3` argument, which is the point from which we want to calculate distance. We'll be using the tower position as the parameter when we call the method, which should be the same as the `targeter` position anyway, but by using the parameter instead of just using the `Targeter` position, we make sure we can call this method to get the enemy closest to any point we want, if we ever need to.

The “lowestDistance” is where we'll store the distance between us and the `Enemy` that, so far, is closest to us. We start it at Infinity to ensure that the first distance we calculate is guaranteed to be set as the lowest.

The “enemyWithLowestDistance” explains itself: whenever we find an enemy whose distance is lower than “lowestDistance,” we'll update “lowestDistance” and store the `Enemy` reference here.

Once we've looped through all enemies and checked their distance, we'll be left with “enemyWithLowestDistance” storing the one that was closest. We then just return that enemy.

The loop starts off like they all do: “i” begins at 0 and increases by 1 until it matches the Count of the List. We declare a local variable “enemy” to store the current enemy, “enemies[i]”.

We then check if the enemy is null, or, if it is not null, we check if it's dead. In some of our first chapters, we briefly mentioned the purpose of the exclamation mark ! which we see here, placed before “enemy.alive”. It just flips the value of the bool: if it was “false” it becomes “true,” and vice versa. It's equivalent to saying “enemy.alive == false” except it makes us look smarter when we type it this way.

We do the same thing as before to remove the enemy, but this time, we decrease “i” by 1 to account for the indexes shifting back, as we just discussed.

If the enemy is not dead, we calculate the distance between them and the given “point” parameter using `Vector3.Distance`. We compare that distance to the lowest distance we’ve found so far. If this one is lower yet, we update our lowest distance and store a reference to that `Enemy`, which will later be returned.

This method will also automatically handle a situation where the method gets called, but there are no enemies in the `List`. It will return `null` instead of throwing an error. The local variable “`enemyWithLowestDistance`” will be initialized to `null`. The `for` loop will do nothing at all if the `List` has a `Count` of 0, so it gets skipped completely. Then, we return that variable, which is still “`null`.”

With that, our `Targeters` are ready for use in towers. We’ll just use them as an empty `GameObject` with a trigger `Sphere` or `Box` collider attached, in the `Targeter` layer. They’ll be children of our root `Tower` `GameObject` so they go wherever the tower goes.

We’ll set one up after we’ve coded our tower logic.

Towers

For this chapter, the only tower we’ll be fully implementing is the `Arrow Tower`. However, we’ll be using inheritance to set up a system where future tower types we create will be able to reuse the portion of functionality that they share with the `arrow tower`.

We went over the towers we expect to implement in the previous chapter: the `Arrow Tower`, `Cannon Tower`, `Barricade`, and `Hot Plate`. Let’s review the scripts we’ll be declaring to resemble those towers and how we’ll get the effects we want for each tower by using those scripts.

Tower

Base class for all towers. It’s a normal script, inheriting from the script class `MonoBehaviour`.

It defines how much the tower costs and how much of that cost is refunded to the player if they were to sell the tower. A mere `Tower` will do nothing to hurt enemies, but will block them, forcing them to walk around it. We’ll use it for barricades, since that’s their only purpose, but all other towers will use lower types inheriting from `Tower` to implement their damage-dealing logic.

TargetingTower

Inherits from `Tower`.

This is a tower with a `Targeter` reference and a `range` variable (which is an `int`), both set in the `Inspector`. The `range` variable directly corresponds to the size of the

Targeter, and we'll set the Targeter to that size in the Start method of the tower (using the SetRange method we declared for Targeter).

We won't use this type directly for any towers, but the Hot Plate tower can inherit from it to use the Targeter as a box collider that detects enemies touching the plate. The Targeter will be sized the same as the tower itself, which will be a thin cube on the ground. Since the hot plate script will inherit from TargetingTower, it can access the Targeter reference we declare, reaching into it to loop through all targets and "burn" them all on every frame.

This means most of our hot plate logic is already handled by our Targeter. All we have to do with the script is make it deal damage.

FiringTower

Inherits from TargetingTower.

This is the script we'll use for arrow towers and cannon towers. The logic of targeting a single enemy that's within range and periodically firing a projectile at them is handled here. We'll automatically assign a new target enemy whenever the targeted enemy dies or gets out of range. The closest enemy within range is targeted whenever we need to find a new target. Of course, we'll use the Targeter reference inherited from TargetingTower to find the closest enemy – remember, we declared a method for that as well.

As far as firing projectiles goes, we'll have a reference to the projectile prefab we want to spawn. All we need to do is spawn it at the tower "projectile spawn point" Transform position and call its Setup method to pass in the damage, speed, and target enemy provided by the tower. The rest is handled by the projectile itself. If it's a seeking projectile (for arrow towers), it will home in on the targeted enemy. If it's an arcing projectile (for cannon towers), it will arc at the initial position of the targeted enemy and impact there.

This specification handles all of our use cases. We've planned out the inheritance in a way that lets us reuse functionality that needs to be shared between towers. If we coded each tower type individually, we'd have to give each one certain members that they all share anyway: how much gold they cost, how much gold they sell back for, the Targeter reference used by all but the barricade, and so on. Instead, we just inherit from the correct type, and those members are automatically shared. Most importantly, any future code that deals with towers now has this base "Tower" class that it can interact with to resemble any tower.

Let's get to it. Create a Tower script and write this code in the script class:

```
public int goldCost = 5;

[Range(0f,1f)]
public float refundFactor = .5f;
```

All we're giving our base towers is a gold cost (how much money we must pay to buy the tower) and a refund factor, which is a float between 0 and 1 resembling what fraction of the goldCost you are paid back when you sell the tower. That is, the user loses "goldCost" money when they buy the tower. When they sell it, they get back "goldCost * refundFactor". So if refundFactor is .5f, then the user only gets back half of the money they spent for the tower. This is a common feature in tower defense games. If we gave the player back all their money whenever they sold a tower, they might as well just pawn back all their cannon towers before facing an air level and then make arrow towers instead, since cannon towers can't attack air enemies. This mechanic fixes that, punishing the player a bit for selling their old towers.

Other than that, we have no scripting to do for a basic tower. They have no functionality in and of themselves.

Moving down the line, let's make a TargetingTower script with this code in it:

```
public class TargetingTower : Tower
{
    public Targeter targeter;
    public int range = 45;

    protected virtual void Start()
    {
        targeter.SetRange(range);
    }
}
```

First, we make sure to inherit from Tower, not MonoBehaviour, in the declaring line of the class, after the colon ":". We declare a Targeter reference and a range. We declare a Start method – again, making it virtual because we want to ensure that lower types are able to override it if they need to. In this method, we call the SetRange method for our Targeter, setting it up with the correct range value as soon as possible.

That's all we need for our TargetingTower.

Arrow Towers

Before we script our arrow tower, let's set the GameObjects up so we know the hierarchy we'll be working with. Towers will always be 10 units wide and 10 units long at the base. They shouldn't exceed this size. This consistency will help us later, and keeps everything looking uniform.

When you're done, the arrow tower will look like Figure 28-4.

- Create an empty GameObject. Name it Arrow Tower. Set its layer to Tower.
- Right-click the Arrow Tower and create a child Cube named Base. Scale it to (10, 6, 10) and set its local position to (0, 3, 0). This will put its bottom at the position of the root Transform, as we've been doing.
- Right-click the Base and create a child Cylinder. Just leave its name as Cylinder. Set its local position to (0, .6, 0) and scale to (.8, .1, .8).
- Right-click the Cylinder and create a child Cube. Name it Barrel (even though it's quite rectangular). Position it at (0, 2.2, .3) and scale it at (.2, 2.5, .65).
- Right-click the Barrel and create an empty GameObject child. Name it Projectile Spawn Point. Position it at (0, 0, .5) and leave its scale as is.
- Create an empty GameObject named Targeter that's a child of the root Arrow Tower. Change its layer to Targeter. Give it a **trigger** Sphere Collider and a **kinematic** Rigidbody.
- Add an instance of our Targeter script to the Targeter. Set the "Col" field to reference the Sphere Collider of the Targeter.
- I'll make a Tower material and give all of the pieces a low-saturation blue color with a hex value of 7698B1.

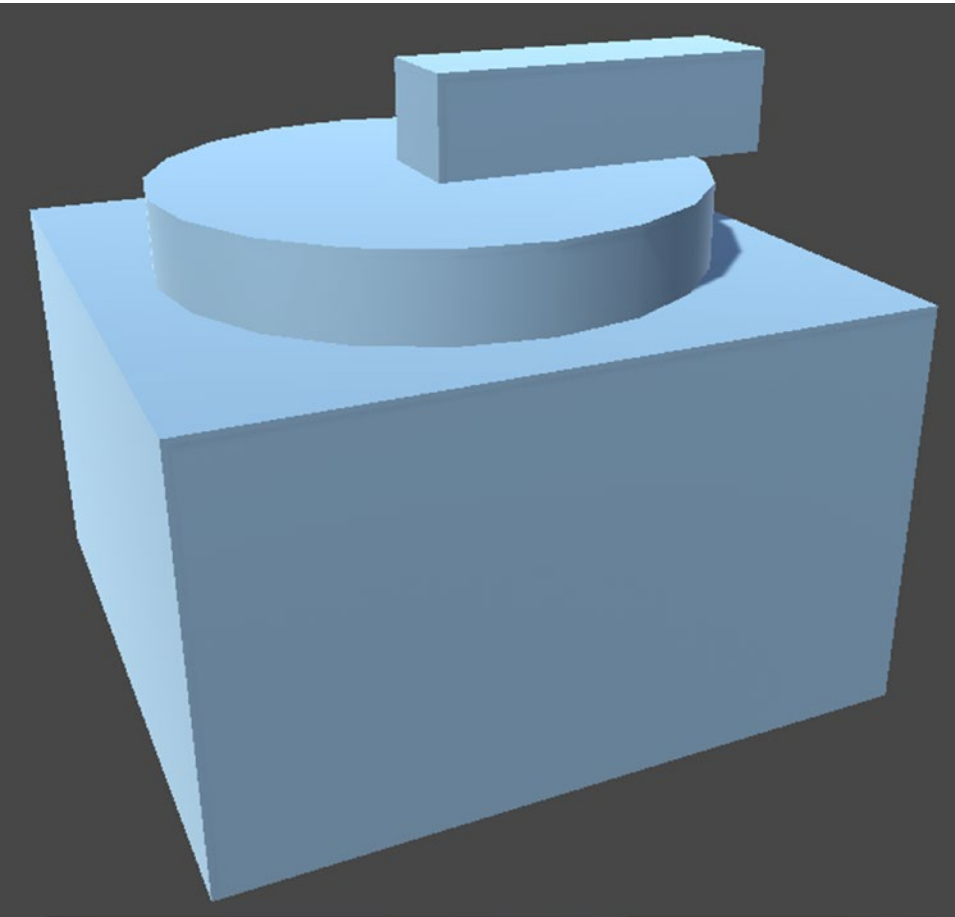


Figure 28-4. *Our Arrow Tower*

Let's move on and create a `FiringTower` script. We'll have to write some actual per-frame code for this!

Change the declaring line to make it inherit from `TargetingTower`:

```
public class FiringTower : TargetingTower
```

Add these variables within the script class:

```
[Tooltip("Quick reference to the root Transform of the tower.")]
public Transform trans;

[Tooltip("Reference to the Transform that the projectile should be
positioned and rotated with initially.")]
public Transform projectileSpawnPoint;
```

```

[Tooltip("Reference to the Transform that should point towards the enemy.")]
public Transform aimer;

[Tooltip("Seconds between each projectile being fired.")]
public float fireInterval = .5f;

[Tooltip("Reference to the projectile prefab that should be fired.")]
public Projectile projectilePrefab;

[Tooltip("Damage dealt by each projectile.")]
public float damage = 4;

[Tooltip("Units per second travel speed for projectiles.")]
public float projectileSpeed = 60;

private Enemy targetedEnemy;

private float lastFireTime = Mathf.NegativeInfinity;

```

You're an elite programmer now, so I'll let you read the tooltips to see what the purpose of each variable is. Even if you think you don't get it, don't fret. We'll be seeing the variables in use soon and going over how they're being used.

The two at the bottom that don't have a tooltip are somewhat self-explanatory: the enemy we're currently targeting, which is the one we'll be shooting at, and the `Time.time` at which we last fired a projectile, which we'll use to know when it's time to fire our next projectile.

Our Update method will use a few separate methods to split the logic up neatly and reuse some of it. Let's start with that, to get a good overview of how it all works:

```

void Update()
{
    if (targetedEnemy != null) //If there is a targeted enemy
    {
        //If the enemy is dead or is not in range anymore, get a new target:
        if (!targetedEnemy.alive || Vector3.Distance(trans.
            position,targetedEnemy.trans.position) > range)
        {
            GetNextTarget();
        }
    }
}

```

```

        else //If the enemy is alive and in range,
        {
            //Aim at the enemy:
            AimAtTarget();

            //Check if it's time to fire again:
            if (Time.time > lastFireTime + fireInterval)
            {
                Fire();
            }
        }
    }
    //Else if there is no targeted enemy and there are targets available
    else if (targeter.TargetsAreAvailable)
        GetNextTarget();
}

```

If there is a targeted enemy set (it's not null), we then proceed to check if they are dead or otherwise outside of the tower's range. This is done with the trusty `Vector3.Distance` call, comparing our transform position to that of the targeted enemy. If the distance exceeds our "range" variable, we can't shoot that enemy anymore, so we call a method `GetNextTarget()`. We'll declare that in a bit – it's just one line of code, calling the `Targeter.GetClosestEnemy` method.

Otherwise, if the enemy is alive and in range, we'll call `AimAtTarget`, which will rotate the portion of our tower that holds "the barrel" toward the target enemy (we'll get to declaring that soon). Then, we check if the current game time has gone over the time at which we last fired, plus the `fireInterval`, which is how long we want to wait between each projectile being fired. We've done this sort of thing before, so that's no big deal. Of course, we're calling another method we have yet to declare, `Fire`. That method will set "lastFireTime" to the current time and spawn the projectile.

Let's declare the `AimAtTarget` method. It's a little bulky, but simple enough:

```

private void AimAtTarget()
{
    //If the 'aimer' has been set, make it look at the enemy on the Y axis
    only:
    if (aimer)

```

```

{
    //Get to and from positions, but set both Y values to 0:
    Vector3 to = targetedEnemy.trans.position;
    to.y = 0;

    Vector3 from = aimer.position;
    from.y = 0;

    //Get desired rotation to look from the 'from' position to the 'to'
    position:
    Quaternion desiredRotation = Quaternion.LookRotation((to - from).
    normalized,Vector3.up);

    //Slerp current rotation towards the desired rotation:
    aimer.rotation = Quaternion.Slerp(aimer.rotation,desiredRotation,.
    08f);
}
}

```

The “aimer” will be set to the Cylinder on top of the tower, which is the parent of the Barrel, so the Barrel will spin with it too. We only want the Cylinder and Barrel to rotate toward our target and only along the Y axis, which spins without tilting it off the tower. Due to the way we set up the barrel, it sticks out along the forward axis of the Cylinder. So as long as we point the cylinder directly toward the target enemy, the barrel will point there as well.

To make sure we’re operating on the Y rotation axis only, we set up these two Vector3 variables, “from” and “to,” so that we can set their Y position to 0. Then we use them to get a direction to point from the “aimer” and toward the target enemy. By leveling their Y axis, we take it out of the equation. As far as the direction is concerned, everything is on the X and Z axes only – they’re always equal on the Y axis, so that the “from” and “to” are never considered higher or lower than the other. This way, we don’t look up or down at enemies, just outward, spinning in a circle atop the tower. That equates to Y rotation axis only.

We use Quaternion.LookRotation to get the direction from “from” toward “to,” as we’ve done before. In case you’ve forgotten, LookRotation returns a rotation where the forward facing is pointing at the direction given as the first parameter, and the second parameter, Vector3.up, is where the up axis should point. So we’re saying “point the front of the

cylinder at the target enemy, and keep the top pointing upward.” If we did `Vector3.down` instead, the top of the cylinder would instead point straight down, which would flip it over.

We store that rotation in the “desiredRotation” variable. Then, we `Slerp` the “aimer” rotation toward that variable at a fraction of `.08f` per frame. This makes it nice and smooth.

Now we need `GetNextTarget`, which is just one line of code – but we need to use it twice, so why not make a method, right?

```
private void GetNextTarget()
{
    targetedEnemy = targeter.GetClosestEnemy(trans.position);
}
```

It just resets our `targetedEnemy` to the enemy that’s closest to the tower. If there is no enemy within range, we’ll get “null” back; and, according to the `Update` code, we’ll wait until there are targets available before trying again.

Now, the most important bit is shooting arrows at the target:

```
private void Fire()
{
    //Mark the time we fired:
    lastFireTime = Time.time;

    //Spawn projectile prefab at spawn point, using spawn point rotation:
    var proj = Instantiate<Projectile>(projectilePrefab,
    projectileSpawnPoint.position,projectileSpawnPoint.rotation);

    //Setup the projectile with damage, speed, and target enemy:
    proj.Setup(damage,projectileSpeed,targetedEnemy);
}
```

We make sure to set the `lastFireTime` to the current `Time.time`; otherwise, we’ll be shooting arrows every single frame. We then spawn the projectile, storing it in a local variable. The first parameter is the prefab to spawn, the second is the position to spawn it at (we use the spawn point `Transform` at the tip of our Barrel), and the third is the rotation it should use (again, the same rotation used by the spawn point `Transform`, aligned with the Barrel).

This might seem a little off to you – we’re Instantiating an entire prefab, but we’re referencing just the `Projectile` script to do it. Remember, the “`projectilePrefab`” variable

is of the `Projectile` type, not `GameObject`. Are we not just Instantiating a lonely `Projectile` script instance? Is that even possible? What would it be attached to?

Well, that's not what we're doing. Unity allows us to Instantiate prefabs and get back a reference to whatever part of them we actually want to interact with. When we Instantiate a script, we're really saying "I want to create this prefab, but when you're done, just return to me the `Projectile` script." This spares us having to call `GetComponent<Projectile>` after Instantiating the prefab.

You can do this with different component types too – say you were coding a first-person game and wanted to create a block and throw it outward on the spot. You could reference the block prefab as its attached `Rigidbody` (instead of as a `GameObject`), then `Instantiate<Rigidbody>`, and use the reference to add force to the `Rigidbody` that throws the block outward. It's just a little more convenient than if we were forced to Instantiate the `GameObject` of the prefab.

And as you can see, the reason we want to access the `Projectile` script is so we can easily call the `Setup` method afterward, passing in the tower damage and projectile speed and giving the projectile its targeted enemy.

That'll do it. Add a `FiringTower` script to the root Transform of our Arrow Tower, go over all the fields in the Inspector, and set the references we'll need (you know what they are by now!); and don't forget to make a prefab out of the root `GameObject`. Figure 28-5 shows how the `FiringTower` should look in the Inspector once you're done.

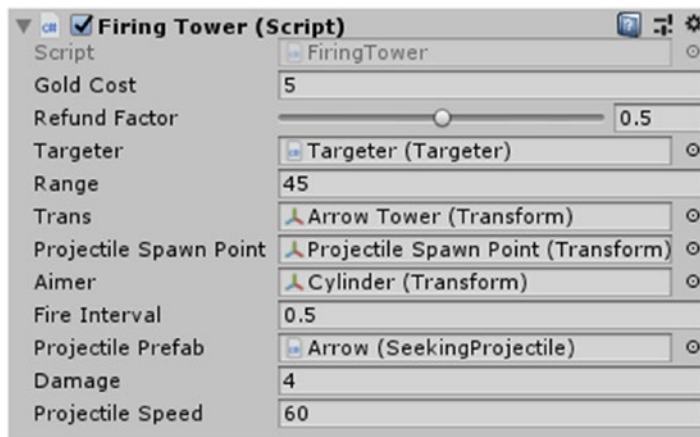


Figure 28-5. A look at the `FiringTower` script of our Arrow Tower in the Inspector, with all the references correctly set

Now you can put it all together and see it in action. Throw some arrow towers down with your prefab, or copy-paste the instances that are already in the scene. The Test

Enemy shouldn't have a prefab made because we'll be making Ground Enemies and Flying Enemies instead, but for the sake of testing, you can copy-paste to create some extra test enemies, put them within range of the towers, and play the game. If everything is set up right, the arrow towers should automatically find their targets and fire at them until they perish.

If your arrow towers aren't firing, make sure you've got everything in the correct layers: test enemies in the Enemy layer, towers in the Tower layer, and the Targeter of the towers in the Targeter layer. If you still experience problems, run through your script instances, like Enemy, FiringTower, and Targeter, and double-check that all of the references are set correctly.

Figure 28-6 shows a setup with enemies and arrow towers in action. I've given the Stage plane a material to darken it so the towers and enemies look nice on it, using a dark-blue color with a hex value of 0A1F38.

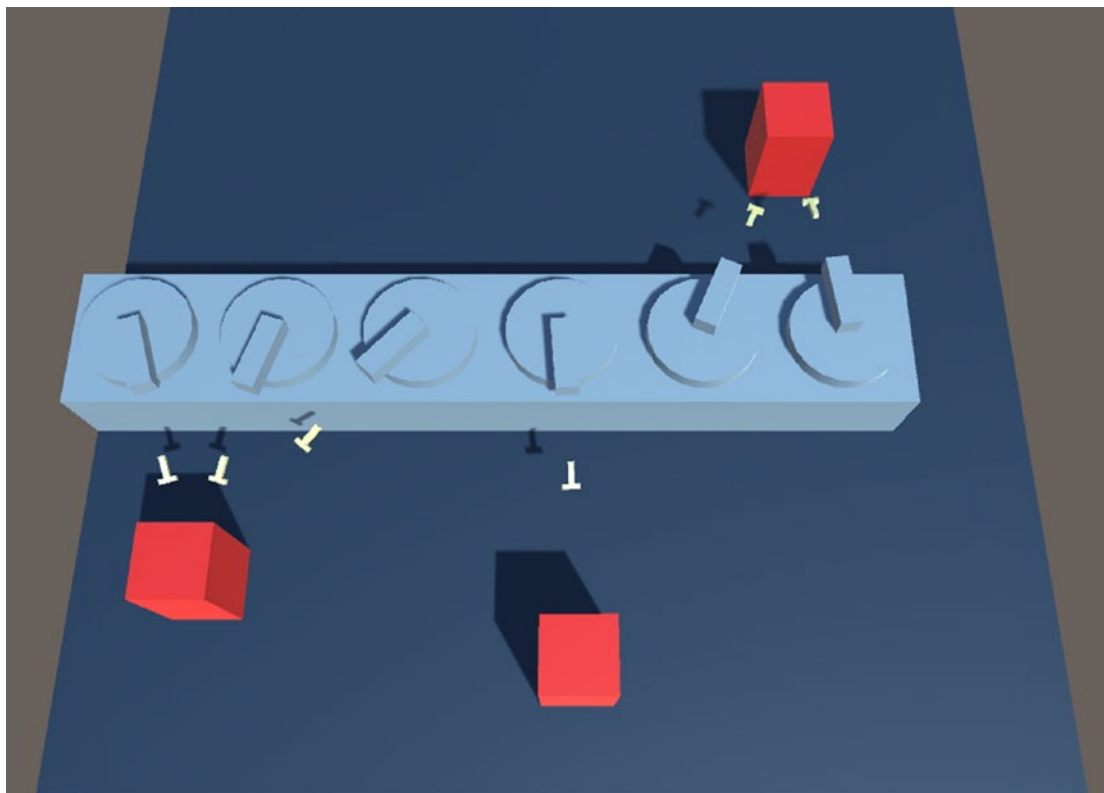


Figure 28-6. A row of six Arrow Towers firing at three test enemies positioned around them

Summary

This chapter implemented our first tower type, the Arrow Tower, and basic enemies that don't yet move. Using inheritance, we've set a solid foundation for implementing the other tower and enemy types. Some points to remember are as follows:

- A class marked as **abstract** is meant to serve as a base class for others to inherit from. You can't create instances of an abstract class. You're expected to inherit from the abstract class and create instances of the inheriting class.
- When removing items from a List within a loop, remember to subtract from "i" to account for the change in indexes when an item is removed.
- A method declared as **virtual** can be **overridden** by lower types to allow inheriting classes to provide their own implementation of the method.
- **Overriding** a virtual method is done by declaring a method with the same name and return type in an inheriting class, but with the "**override**" keyword before the return type.
- A method declared as **abstract** is like a virtual method, but it has no implementation of its own (no code block after the declaration), and it must be part of an abstract class. Any inheriting classes must declare their own **override** version of the abstract method, or an error will be thrown.

CHAPTER 29

Build Mode

In this chapter, we'll implement the user interface that shows during "build mode" to allow the user to buy new towers and sell old towers. Before we begin, let's outline the functionality we're going to implement. From the player's perspective, what will they be able to do in build mode? After this, we'll do a quick primer of the important components and concepts behind Unity's UI. We'll then set up our own UI and learn how to make our buttons actually do something when we click them, implementing all of the build mode functionality bit by bit.

Our finished Build Mode UI is shown in Figure 29-1. We'll be using Unity's new UI system to design our UI out of GameObjects with appropriate components rather than coding using the GUI or GUILayout method like we did before. Since the UI is a bigger part of the game this time, we'll put more effort into making it passable without creating any art for it ourselves. However, to keep the project from becoming unwieldy, we won't be making UI for a start screen or an in-game pause menu or anything of that sort. We did that in the first example project.

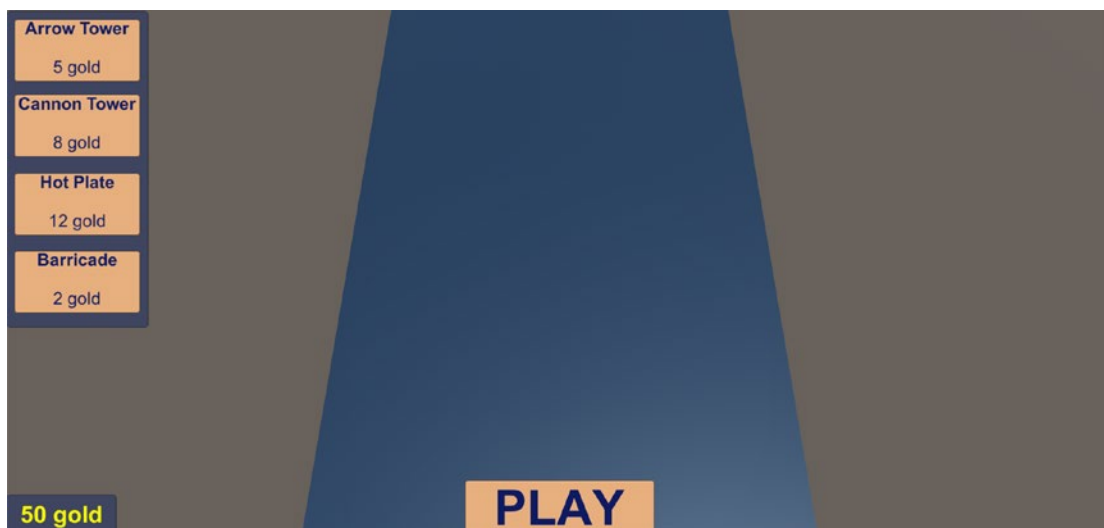


Figure 29-1. End result of our Build Mode user interface

The buttons on the left side are what we'll call the **build buttons**. Clicking any of these buttons will select the corresponding tower for building, and the button will change color to indicate that it is the currently selected button. One build button can be selected at a time, or none at all can be selected.

While a build button is selected, you can left-click the stage to attempt to build the tower at that location. Pressing Escape will deselect the button.

While there is no build button selected, you can instead left-click the stage to select an existing tower. This will show a little popup panel that provides us with an option to sell the tower. This panel will also have a little X button in its top-right corner that you can press to close the panel if you don't want to sell the tower – or you can press Escape. To continuously position this panel over the tower no matter how the camera is positioned, we'll learn how to convert a world position to its corresponding location on the screen – in other words, go from “world position” to “screen position.” This way we can keep updating the location of the UI so that it always draws above the Tower.

Whenever a tower is bought, we'll use Unity's built-in pathfinding to check if the way is clear from the enemy spawn point at the upper end of the stage to the enemy leak point at the bottom end of the stage. If the way is not clear, we won't let the player start the level until they sell something to clear the path.

Build mode will be the initial state of the game when the player first starts playing. They'll start with some gold to spend on their first towers, and when they're ready, they'll hit Play. The enemies will spawn and move toward the leak point. Once all enemies are gone, we'll enter build mode again, and the cycle repeats.

In this chapter, we'll set up the Play button and define an empty method for performing the pathfinding. In the next chapter, we'll actually implement the pathfinding.

UI Basics

One could write an entire book about designing and implementing UI in Unity. I'm sure there are more than a few out there. This chapter won't go into that much detail, because that would be a very long chapter. We'll go over the basics of Unity's UI system and put it to use. When we're done, our UI will be mapped out, and the build mode features will be functional.

There are some considerations to make when you go about designing and implementing a user interface. You must ensure that the placement of elements is conducive to differing screen sizes and ratios.

You might already know this, but a screen is made up of many small dots called pixels. Each dot is a tiny colored light on your screen. These many thousands of dots spread over the monitor will together form the picture that we see. The resolution of a screen is its width and height in pixels (that might not be the exact definition of the word, but pretty much everyone on Earth uses it that way).

Since different screens sometimes come in different sizes, we can't be sure just how much space we have to work with. Some monitors have a differing aspect ratio. That's the width-to-height ratio of a monitor. Common HD monitors are 16:9, which means 16 pixels wide for every 9 pixels tall. Common resolutions for this aspect ratio are 1600×900 and 1920×1080. But your user could be running the game on a different aspect ratio, like 16:10, 5:4, 4:3, or some new ratio or blown-up resolution invented ten years down the road. You can't rule out that a player might have any of these aspect ratios.

To combat this, we have certain tools for positioning and sizing elements based on the screen width and height to ensure that there aren't any situations where things just don't show up on your screen. If you design your UI to have a constant, static size of 1600x900, any lower resolution will simply cut off part of your UI because the screen can't contain it.

We'll learn about Unity's solutions for this in a bit.

Unity's UI system represents each element of the UI as a **GameObject** with relevant UI components attached, depending upon the type of element. These elements can be found through **GameObject ► UI**. We start with a **Canvas** **GameObject**, which is the root of all of our UI. All UI elements will be children of that **GameObject**. If you create a UI element without a Canvas in your scene, a new Canvas will be created, and the element you made will automatically be made a child of the new Canvas.

The Canvas represents the area that the UI is drawn in. Once you've created a Canvas, you'll see a white rectangle showing in your scene that resembles the Canvas. This is easiest to spot when you switch to 2D mode by clicking the button just under the top-left corner of the Scene window (beneath the tab that reads Scene), shown in Figure 29-2. It's also recommended to keep your Tool Handle Position and Rotation at Pivot and Local, shown also in Figure 29-2 (toggle these with the Z and X hotkeys).



Figure 29-2. The Scene tab with the 2D button toggled on, the Tool Handle Position set to Pivot, and the Tool Handle Rotation set to Local

Once you're in 2D mode, the Scene window will not allow you to use the Z axis, instead restricting you to the X and Y axes only, looking forward into the scene. This is ideal for editing UI, since it keeps a straight-on look at the Canvas. Similarly, if we were working on a 2D game, we'd keep this setting on all the time.

Select the Canvas in the Hierarchy and press F to center your view on it. That white rectangle is the bounding box for all your UI. Any UI positioned outside of it will not be visible. If you create a UI element, like a button, you can then switch to your Game window to see the button displaying on the screen, as long as the button is within that white rectangle.

When a Canvas is created, Unity will also create a new GameObject called "EventSystem" if one does not already exist. This handles input for interactable elements like buttons. If all of your elements stop responding to your input, you may have accidentally deleted this GameObject. You can add a new one through GameObject ► UI ► Event System. There should only be one in your scene.

Since UI elements are GameObjects, their parenting will "attach" them to each other such that moving a parent will move its children as well. It also determines the order that elements are rendered in. Elements are drawn in a top-to-bottom order: those higher in the hierarchy will be drawn first. This means if you have two elements that overlap each other, the one that's lowest in the Hierarchy will appear "on top" of the other, covering it.

You can change the order by dragging and dropping in the Hierarchy, just like with normal GameObjects. There are also methods in the Transform class that allow you to change order by code, if you should ever need to.

The Canvas GameObject has components that deal with the way the UI renders and scales with screen size.

The **Render Mode** field of the Canvas component has three possible settings that determine how the canvas and its elements are rendered:

- **Screen Space – Overlay** will render elements over the screen with no fancy side effects. It's your average UI setup.
- **Screen Space – Camera** is much like Overlay, but it references a target Camera and renders the UI as if viewed by that camera. You can use this to give the UI a perspective tilt.
- **World Space** considers elements to be part of the world. This can be used to create interfaces that are meant to seem as if they are in the game world itself, such as a floating hologram menu or menus on an in-game computer screen.

The **Canvas Scaler** component handles the screen size and how the elements adapt to changes in the screen size. Most notable is the **UI Scale Mode** field, which has three possible options:

- **Constant Pixel Size** will keep every element the same size in pixels, regardless of the size of the screen or the aspect ratio of the screen. This can cause elements to appear relatively large on small screens compared to large screens or vice versa.
- **Scale With Screen Size** will provide us with a **Reference Resolution** field (a width and a height). This is the size the UI is designed for. If the screen is larger or smaller, the UI is scaled up or down proportionately. It also provides us with a **Screen Match Mode** field determining how the scaling reacts if the aspect ratio of the screen is different than that of our Reference Resolution. There are three options for this field:
 - **Match Width or Height** will scale all elements by the difference in the width, the height, or a combination somewhere between. A resulting Match field between 0 and 1 determines how much a change in width will affect our UI sizing vs. how much a change in height will affect it. A value of 0 will cause the UI to only scale if the width is changed. A value of 1 will only scale if the height is changed. A value of .5 will provide an even mix of both.
 - **Expand** will increase the size of the canvas, but never decrease it from the reference resolution.
 - **Shrink** will decrease the size of the canvas, but never increase it from the reference resolution.
- **Constant Physical Size** will size UI elements by their physical size rather than a number of pixels. We specify a physical unit measurement to use, such as inches, centimeters, or millimeters, and the width and height of our elements will use this measurement instead of pixels.

The RectTransform

Rather than the Transform component that all other GameObjects use, UI elements will have a different, lower type of Transform called **RectTransform**. As a lower type, it has all of the members of a Transform, plus some of its own. Most notably, it adds a width, height, pivot point, and anchoring. The width and height work in conjunction with scale, acting not as a multiplier like scale, but a float value depicting the size.

You'll most commonly move your elements using the rect tool (hotkey T). We've used it before, so you're already familiar with it – but note that when you change the size of an element by dragging at its edges or corners, the width and height is what you're changing, not the scale. However, the scale tool (hotkey R) still changes the scale of the element, leaving the width and height unaffected.

This difference is most notable if you ever change the size of an element that has children inside it. Scale will affect all children, while the width and height do not.

The pivot point of a RectTransform is resembled by the circle icon, by default in the center of the element. To change the pivot, click and drag the circle. It will snap to the edges of the element and the center of the width or height. Notice that rotation will spin the object around the pivot point.

The RectTransform employs a concept of **anchors**, which provide a means of fixing an element in a specific place relative to its parent element.

The anchors are visualized by four white triangular handles. By default for a new element, they'll all be positioned in the center of the Canvas, bundled up together, shown on the left side of Figure 29-3. You can click and drag the center of them to move them all at once or click an individual handle to pull it and separate it from the rest, shown on the right side of Figure 29-3. While pulling a single handle, two other handles react, keeping themselves aligned so that the four handles are always making a rectangular shape together.



Figure 29-3. The four anchor handles for a UI element, all positioned together on the left, and the same four handles with some space between them on the right

If you hold Ctrl before clicking an individual anchor handle, dragging will instead move all of them at once.

Each anchor corresponds to a corner of the associated element and how that corner is positioned within its parent element. The corner of the element will retain its position relative to the anchor point.

One common use for this is to anchor elements to one corner of the Canvas. If you leave your anchors at the center of the screen but place your elements at the edges of the screen, a change in screen size will likely mean your elements are no longer on the edge of the screen. To fix this, you can anchor them to the side of the screen you want them attached to. Our panel for build buttons in Figure 29-1, for example, is anchored to the top-left corner of the screen so that no matter the size or orientation of the screen, the elements are always tied to that corner.

The Inspector has a handy means of easily assigning anchor presets, shown in Figure 29-4. This provides various options for assigning the horizontal and vertical anchors to the center or corners of the parent with a single click.

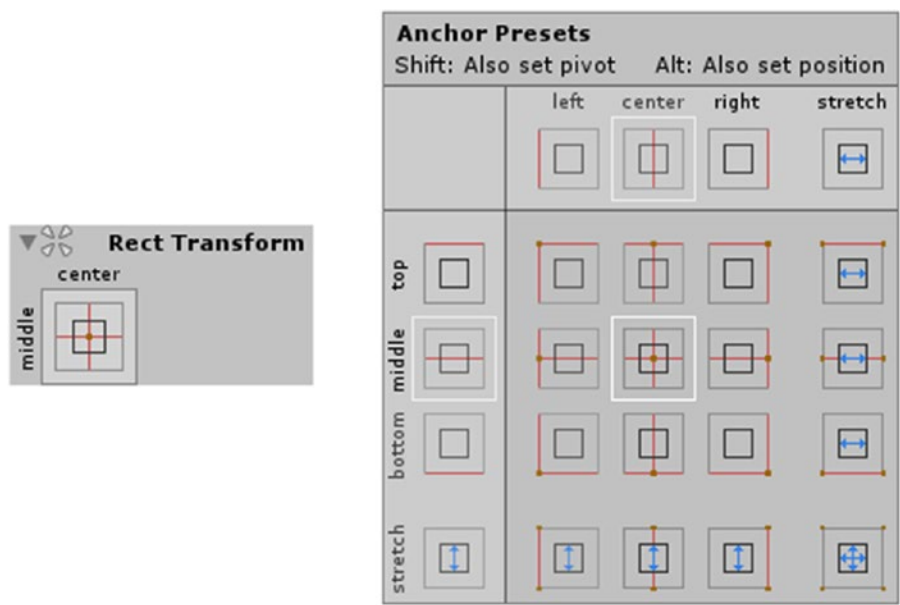


Figure 29-4. The anchor preset button in the Rect Transform of the Inspector (left). When clicked, it shows the dropdown box of anchor presets (right)

Building Our UI

With all that preliminary prattling out of the way, let's get back to the real world and start working on our own UI. Our UI is a simple mixture of panels (colored rectangles) and buttons with text inside them. Since elements are just GameObjects with certain components attached, we can make use of prefabs to define a consistent style that we can change all at once down the road if we want to. We can override the stuff we need to, like what the text says in a button, the size, the anchors, and so on, but keep things like color attached to the prefabs so we can easily make edits to all instances.

Before we begin, let's set up our Canvas, and don't forget to make sure you have an `EventSystem` in the scene:

- Select your Canvas. In the Inspector, locate the Canvas Scaler.
- Change the UI Scale Mode field to Scale With Screen Size.
- Set the Reference Resolution to 1280×720. That's 1280 on the X axis and 720 on the Y axis.
- Set Screen Match Mode to Match Width or Height.
- Change the Match slider to .5.

Now let's create a **generic button**:

- Make a GameObject ► UI ► Button and view it in the Inspector.
- Ignore the Image component and focus instead on the Button component. Change its Normal Color field to an orange with a hex value of F0B683.
- Change its Highlighted Color to FFD6B2, a slightly paler variant. This will show when the mouse hovers over the button.
- Change its Pressed Color to a slightly darker variant with a hex value of DA9E69. This color shows while the mouse button is held down over the button.
- Make its Selected Color match the Normal Color.
- Select the Text element within and set its Color field to a hex value of 0F1B64.

- If you'd like to keep things a little more organized, create a UI subfolder inside your Prefabs folder in the Project window. Drag and drop the Button here to create a prefab for it.

That will serve as the base for our buttons. The text inside isn't important; we'll override that when we implement the prefab. We just want to create a consistent template of colors so we don't have to set the same fields each time.

Moving on, let's make a **generic panel**:

- Create a GameObject ► UI ► Panel. By default, it will be sized to the whole canvas.
- In the Image component, give it a dark-blue color with a hex value of 4B5374. Set the "A" field to 100 if it is not already. This is alpha – a value of 100 makes the color fully solid, while a value of 0 makes it totally transparent. We want it solid.
- Create a prefab for this panel as well. It's just a colored rectangle that we can lay elements on to group them up nicely.

We can use this instance of panel for the **build buttons** on the left side. Using the anchor preset in the Inspector, select the top-left preset, which will anchor the element to the top-left corner of the canvas. Its width and height should show in the Inspector after this. Change these to 180 width and 400 height. Then, apply the same top-left anchor preset again, but this time, hold Alt before you click the preset, which will snap the panel to the top-left. Alternatively, you can use the rect tool, click within the panel, and drag to move it until its top-left corner is at the top-left corner of the Canvas. It should snap into place once you get close enough. While you're at it, set the name to "Build Button Panel."

We want our build buttons to be a bit different than a normal button, since we need a tower gold cost displayed in them as well. We'll create a prefab variant of the Button to make these:

- Right-click the Button prefab in the Project and select Create ► Prefab Variant. Name the variant Build Button.
- Create an instance of the Build Button by dragging and dropping it from the Project onto the Build Button Panel in the Hierarchy.
- Change its width to 160 and height to 80.

- Change the button anchor to the top-left preset.
- Change the name of the Text child element to Tower Text.
- In the Text component, set the Font Style to Bold and the Font Size to 22.
- With the Alignment field, change the alignment of the text by clicking the fourth button from the left. This will move the text to the top of the button.
- The Color field of the text should still be a hex value of 0F1B64 from the base button prefab.
- Copy and paste the Tower Name GameObject and change the name to Gold Cost. Change the Font Style back to Normal and the Alignment field to Center (the second button from the right side). Change the text to “XX gold,” and we’ll fill in the gold cost on a per-tower basis. Using the position tool (hotkey W), move the copy down so it’s near the bottom of the button.
- Select the base Build Button GameObject. Using the Overrides dropdown near the top-right corner of the Inspector, click Apply All to apply the changes we made to our prefab variant.

Now that the generic build button is set up as a prefab, copy and paste it and drag each one down to fill the panel. Sized as they are, you should fit four of them – one for each tower. Just drag them on the Y axis with the move tool (hotkey W) and keep the space between them roughly equal.

Change each button’s GameObject name, for example, “Arrow Tower Button,” if you want to keep things neat-looking. Set the tower name text and the gold cost text accordingly:

- An **Arrow Tower** will cost **5 gold**.
- A **Cannon Tower** will cost **8 gold**.
- A **Hot Plate** will cost **12 gold**.
- A **Barricade** will cost **2 gold**.

With that panel done, let's add a panel that shows our **current gold**. We'll script it to function correctly later:

- Add an instance of the Panel prefab as a child to the Canvas. Name it Current Gold Panel.
- Set its anchor preset to Bottom Left.
- Size it to 140 width and 52 height.
- Right-click the panel in the Hierarchy and select UI ► Text to add a Text element child. Name it Current Gold Text.
- Color the text a pure yellow (the color of sweet gold), hex value FFFE00.
- Make the text bold, give it 30 font size, and write "50 gold". That's how much gold the player will start with. In the Alignment field, center and justify the text by selecting the middle button of each group (second from the left and second from the right).

Now we'll create the **Play button** in the center of the screen:

- Drag and drop an instance of the Button prefab onto the Canvas GameObject in the Hierarchy. Name it Play Button.
- Set its anchor preset to Bottom Center.
- Set its size to 240 width and 70 height.
- Position it at the bottom and center of the screen with the rect tool or hold Alt and reapply the anchor preset to snap it in place.
- Change the text Font Style to Bold and size to 62; center and justify the alignment.

Lastly, we'll create an element that doesn't show all the time, **the Tower Selling Panel**, which we'll position over the selected tower to show a button letting the user sell it, as well as an X button to deselect the tower:

- Create an instance of the Panel prefab as a child to the Canvas. Name it Tower Selling Panel.
- Set its anchor to Middle Center. Using the rect tool, drag its pivot to the middle of the bottom edge. It should snap in place.

- Set its width to 186 and height to 68.
- Override its background color to a lighter blue with a hex value of 8BB0D8.
- Add an instance of the Button prefab as a child to the Tower Selling Panel. Name it Sell Button and give it a width of 110 and a height of 58. Set its X position to -32.
- Rename the text GameObject of the Sell Button to SELL Text, change the text to “SELL,” make it bold, give it a font size of 32, and justify the text at the top with the Alignment field.
- Copy-paste the SELL text; rename it to Refund Text; set its text to “for XX gold”; set its Font Size to 18, Font Style to Normal, and Alignment to justify and center; and drag it down with the move tool until it’s under the “SELL” text.
- Add another Button instance to the Tower Selling Panel. Name it “X Button”. Leave its anchor at Middle Center and set both its width and height to 38. Set its X position to 60 and its Y position to 10.
- Set the X Button text to read “X”. Make it bold, justified, and centered with a font size of 34.
- Finally, deactivate the Tower Selling Panel. We’ll make it show by script when we need it to, but we don’t want it to show by default.

The Refund Text will be referenced in our Player script, and we will update the text to replace the “XX” with the actual refund value of the selected tower whenever the player selects a new tower. When you’re done, the Tower Selling Panel should look like Figure 29-5, and the other elements we’ve created should look like Figure 29-1.



Figure 29-5. The Tower Selling Panel

Events

UI elements that have some form of interaction with the user will have event fields exposed in the Inspector, such as a button having an “OnClick” event. These fields allow us to attach functionality to the event so that some action is performed when the event happens – like when a button is clicked. We’ll learn how to do it ourselves in a moment, but first, let’s go over the concept.

We can add as many actions as we want to a single event for a single UI element. Each action we add will first ask us to reference some object we want to interact with. Then, a dropdown field will appear that lets us point to some member on that object: a variable or a method.

If we drop a `GameObject` on this field, we can access members from any of the components attached to it or from the `GameObject` itself. For example, if we drop the `Player Camera GameObject` on the field, we can access members from the `GameObject` type as well as the `Transform` and `Player` script.

What happens next depends on what we point to within the object we referenced:

- **If we point at a variable**, a field will pop up that lets us set the value of the variable. When the event occurs, the variable value is set to whatever we put in that field.

- **If we point at a method**, it must be a public method with no more than one parameter, and the parameter must be of a type that's serializable – such as a basic value type, a script, or a built-in component. If the method declares a parameter, we'll get a field to set the parameter value. When the event occurs, the method is called, using the parameter value we provided in the field.

The type of data accepted in this field will change based on the type the code expects, of course – a string variable will show a field you can type in, a `GameObject` variable would show a field to reference a `GameObject` by dragging and dropping from the Hierarchy, and so on.

We can't reference variables that we declare in our own scripts, but we can reference those of built-in Unity types. The Object data type accepted by the field is the base class for many Unity types, so we can reference a `GameObject` and set its "name" variable to change its name, or we could call its `SetActive` method to deactivate or activate it. We could reference a component and call one of its built-in methods. It doesn't have to be code we wrote ourselves. But it can be that too – so long as we call a method we declared, rather than trying to set a variable.

We'll use these events to call methods from our Player script when it's time to implement the functionality of our buttons. We'll give each build button method calls that set the corresponding Tower prefab to build and give us a reference to the button so we can change its color back and forth when it's selected and deselected.

When our script is ready to build a tower, it can use that tower prefab variable to determine which tower to Instantiate, and since it's a Tower, it will have a `goldCost` variable we can use to determine if the player has enough money to build it.

But enough talking about it. Let's get to writing code to make our UI more than just a bunch of buttons that do nothing.

Setting Up

When we place towers on the stage, we'll always keep them at a Y position of 0, with X and Z values at multiples of 10 (e.g., 10, 20, 30, and so on, including negative values). A tower is 10 units wide and long, so we're only allowing the player to place them along increments of the tower size. Ultimately, it's something like a grid of towers, where only

one tower can be in a single cell of the grid at a time. In order to place a tower on the stage, we need to let the user point and click where they want it to go, though.

To do this, we'll be exploring a handful of new methods and concepts. We'll learn how to perform a raycast. This is a physics method that effectively "shoots a ray" out into the scene to test if hits any colliders. You define a point for the ray to start from, a direction for it to travel in, and a total distance for it to travel. If the ray hits anything, the method will give back information about what it hit.

To get the point on the stage that the player mouse cursor is hovering over, we can cast a ray starting at the mouse cursor position and shooting out at the stage. Conveniently, Unity has a built-in method in the Camera component that lets us convert a point on the screen to a Ray instance. The Ray is a type that stores an origin and direction of a ray, which we can plug into the raycast method.

Let's dive in. We'll start by giving the player an indication of where their mouse cursor is hovering over the stage. To do this, we'll raycast from the mouse and get the point the ray touched on the stage and then position a "highlighter" object there. But we don't want to just show the player the exact point on the stage that they touched. We want it to conform to multiples of 10 to show them the sort of grid pattern that towers will be placed in.

To start, we'll make our highlighter be a thin cube the size of a tower:

- Create a Cube with no parent. Name it Highlighter.
- Set its scale to (10, .4, 10).
- Create a material named Highlighter and give the cube a color with a hex value of B7FAF6. We'll also make it a bit transparent. Give it an alpha of 30 and change the Rendering Mode field at the very top of the material in the Inspector to "Transparent." If you leave the Rendering Mode at the default setting of "Opaque," the alpha will be ignored, and the object will always be fully solid.
- By default, deactivate the highlighter.

This cube is now the size of a tower on the X and Z axes, so we just need to script it to position itself under the mouse cursor on the stage and to always snap to multiples of 10 with its X and Z position.

We'll use our Player script for most of the build mode functionality, so open it up in your code editor. First, let's create a distinction between build mode and play mode using an enum `Mode`, declared at the top of the Player script:

```
private enum Mode
{
    Build,
    Play
}

private Mode mode = Mode.Build;
```

We'll need to reference the highlighter Transform so we can move it around. While we're at it, let's declare the rest of the variables we'll need for build mode logic. We need to access some classes from the `UnityEngine.UI` namespace, which is where built-in UI components are declared, so add this "using" to the top of the Player script file:

```
using UnityEngine.UI;
```

Then let's declare our variables:

```
[Header("Build Mode")]
[Tooltip("Layer mask for highlighter raycasting. Should include the layer of the stage.")]
public LayerMask stageLayerMask;

[Tooltip("Reference to the Transform of the Highlighter GameObject.")]
public Transform highlighter;

[Tooltip("Reference to the Tower Selling Panel.")]
public RectTransform towerSellingPanel;

[Tooltip("Reference to the Text component of the Refund Text in the Tower Selling Panel.")]
public Text sellRefundText;

[Tooltip("Reference to the Text component of the current gold text in the bottom-left corner of the UI.")]
public Text currentGoldText;
```

```
[Tooltip("The color to apply to the selected build button.")]
public Color selectedBuildButtonColor = new Color(.2f,.8f,.2f);

//Mouse position at the last frame.
private Vector3 lastMousePosition;

//Current gold the last time we checked.
private int goldLastFrame;

//True if the cursor is over the stage right now, false if not.
private bool cursorIsOverStage = false;

//Reference to the Tower prefab selected by the build button.
private Tower towerPrefabToBuild = null;

//Reference to the currently selected build button Image component.
private Image selectedBuildButtonImage = null;

//Currently selected Tower instance, if any.
private Tower selectedTower = null;
```

Let's go over these variables and how we plan to use them:

- **stageLayerMask** – A LayerMask is a built-in Unity type we haven't used before. It resembles a set of all of our collision layers (defined in our Project Settings) and allows us to check or uncheck each one individually. We'll pass it to our raycast call to define which layers the ray should collide with and which it should ignore. In the Inspector, the layer mask field shows as a dropdown listing all layers, where each layer can be clicked to toggle it on or off.
- **highlighter** – A reference to the Highlighter, which we'll use to position it. We'll also go through this to get the GameObject so we can deactivate the highlighter when the mouse is not hovering over the stage.
- **towerSellingPanel** – A reference to the RectTransform of the Tower Selling Panel. We'll use this to constantly position the panel above the selected tower.

- **sellRefundText** – Through a reference to the Text component of our “for XX gold” text element in the Tower Selling Panel, we’ll change the “XX” to match the gold the Tower will give when sold.
- **currentGoldText** – Like the sell refund text, we use this to set the indicator of the player’s current gold in the bottom-left corner.
- **selectedBuildButtonColor** – The color applied to the selected build button to distinguish it as the currently selected button. We initialize this to a default value of 20% red, 80% green, and 20% blue. We can change it in the Inspector if we want.
- **lastMousePosition** – We keep track of the mouse position each frame here. By comparing the mouse position from the last frame to the current mouse position, we can tell if the mouse moved. We’ll only perform our raycast if it’s moved, just to save some unnecessary processing.
- **goldLastFrame** – We also keep track of the gold we had on the last frame, using this to detect when we need to update the currentGoldText.
- **cursorIsOverStage** – If the raycast ever fails to find the stage, we’ll set this to “false” and deactivate our highlighter so it stops showing up.
- **towerPrefabToBuild** – A pointer to the Tower prefab we want to build. This will be set by a method we’ll set to be called from the build button OnClick event. If it’s not null, then a build button is selected, and clicking the stage will attempt to build this Tower.
- **selectedBuildButtonImage** – A pointer to the Image component of the build button that’s currently selected. Again, this is set by the OnClick event we’ll be setting up. We’ll use this to change the color of the selected build button to distinguish it from the others.
- **selectedTower** – Currently selected Tower instance. This is what will be sold when the Tower Selling Panel “SELL” button is clicked. If it’s null, there is no selected tower. This should always be null if there’s a build button selected.

Once you’ve added the code and saved the script, go ahead and set up the references in the Inspector. Once you’re done, the Build Mode variables of your Player script in the Inspector should look like Figure 29-6.

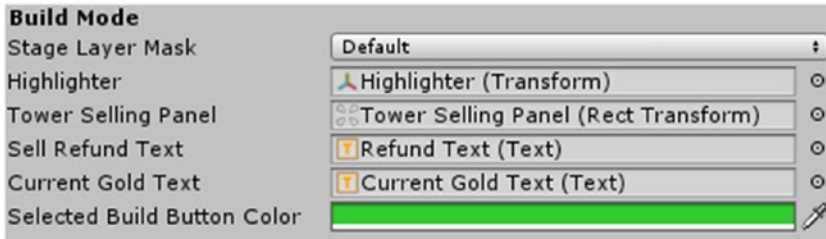


Figure 29-6. The Build Mode settings of our Player script in the Inspector after all references have been set

In the layer mask, we’ve unchecked every layer except for the “Default,” which is the layer our stage will be part of. This is important! If you forget to set a layer mask, it will default to “Nothing,” where no layers are checked. This will guarantee that your raycast won’t collide with anything and probably leave you desperately wondering why.

Build Mode Logic

Let’s declare a separate method to handle all build mode–related logic. In your Update method, add the two lines at the end to call the method only while “mode” is set to Mode.Build:

```
void Update()
{
    ArrowKeyMovement();

    MouseDragMovement();

    Zooming();

    MoveTowardsTarget();

    //Run build mode logic if we're in build mode:
    if (mode == Mode.Build)
        BuildModeLogic();
}
```

And let's declare the method – like our Update method, we'll separate the functionality of our build mode into several other method calls to keep things tidy:

```
void BuildModeLogic()
{
    PositionHighlighter();
    PositionSellPanel();
    UpdateCurrentGold();

    //If the left mouse button is clicked while the cursor is over the
    stage:
    if (cursorIsOverStage && Input.GetMouseButtonDown(0))
    {
        OnStageClicked();
    }

    //If Escape is pressed:
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        DeselectTower();

        DeselectBuildButton();
    }
}
```

Of course, we still have to define these methods we're calling, but this shows us the high-level overview of build mode logic:

- **PositionHighlighter** will raycast from the mouse cursor toward the stage. If it hits the stage, we'll update the highlighter position and ensure that it's active (visible). If it does not hit the stage, we'll deactivate the highlighter.
- **PositionSellPanel** will reposition the Tower Selling Panel to the screen position of the selected Tower, if there is presently a selected Tower.
- **UpdateCurrentGold** will update the text of the gold indicator at the bottom-left corner to match the gold the player currently has.

- **OnStageClicked** will be called whenever the stage is clicked. This uses the “cursorIsOverStage” bool we declared to only call the method when the cursor is actually hovering over the stage. That’s set in the PositionHighlighter method, which performs the raycast.
- **DeselectTower** clears out the selected tower and deactivates (hides) the Tower Selling Panel. We call it on various other occasions.
- **DeselectBuildButton** clears the selected tower prefab and reverts the color of the selected build button back to its normal state.

Let’s start declaring these methods one at a time. First and foremost, let’s position the highlighter and learn how to perform a raycast:

```
void PositionHighlighter()
{
    //If the mouse position this frame is different than last frame:
    if (Input.mousePosition != lastMousePosition)
    {
        //Get a ray at the mouse position, shooting out of the camera:
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit; //Information on what was hit will be stored here

        //Cast the ray and check if it hit anything, using our layer mask:
        if (Physics.Raycast(ray,out hit,Mathf.Infinity,stageLayerMask.value))
        {
            //If it did hit something, use hit.point to get the location it
            hit:
            Vector3 point = hit.point;

            //Round the X and Z values to multiples of 10:
            point.x = Mathf.Round(hit.point.x * .1f) * 10;
            point.z = Mathf.Round(hit.point.z * .1f) * 10;

            //Clamp Z between -80 and 80 to prevent sticking over the edge
            of the stage:
            point.z = Mathf.Clamp(point.z,-80,80);

            //Ensure Y is always 0:
            point.y = .2f;
```

```

        //Make sure the highlighter is active (visible) and set its
        position:
        highlighter.position = point;
        highlighter.gameObject.SetActive(true);
        cursorIsOverStage = true;
    }
    else //If the ray didn't hit anything,
    {
        //... mark cursorIsOverStage as false:
        cursorIsOverStage = false;

        //Deactivate the highlighter GameObject so it no longer shows:
        highlighter.gameObject.SetActive(false);
    }
}

//Make sure we keep track of the mouse position this frame:
lastMousePosition = Input.mousePosition;
}

```

We only perform the raycast if the mouse position is different on this frame than it was on the last frame.

We declare a Ray variable. To get the camera that's tagged "MainCamera" in the scene, we reference Camera.main. The camera included in the scene by default will have this tag, but if not, it should be set in the Inspector, just beneath the name field for the Player Camera, shown in Figure 29-7.



Figure 29-7. The tag field in the head of the Inspector for our Player Camera GameObject

Using the Camera.main member gives us an easy way to grab this Camera component on the fly. The Camera method we call, ScreenPointToRay, returns a new Ray instance that resembles a ray shooting out of the camera, originating at a specific

screen position. The parameter we give is the screen position we want to use. We get a Ray that starts at the mouse position and shoots out of the camera.

After this, we declare another local variable: a RaycastHit named “hit”. We then call Physics.Raycast, and as you can see, we provide the “hit” as a parameter, but we use a new keyword before it: “out”.

A parameter in a method can be declared as an “out” parameter, which means that the value is passed in as a reference, so that any modifications made to it will be reflected in our variable.

To understand what this means, you need to realize the distinction between **value types** and **reference types**.

A **reference type** is created by declaring a **class**. When we reference a class, we are pointing at a single instance of the type. A variable and a parameter will both point to the same instance, and any modifications made to either one will automatically affect the other – because, again, they’re both pointing at the exact same data. If you create a class instance and store it in variable A, then create many other variables that all reference variable A, then everything points to the same, single instance of the class.

A **value type** is created by declaring a **struct**, which is much like a class except that it is treated differently when passed around. Whenever a struct is referenced, such as when assigning a variable or passing a variable into a method call as a parameter, the data is copied and a new instance is given. If you create a struct instance and store it in variable A, then create many other variables that all reference variable A, each variable will store its own instance of the struct. Each assignment copies the struct.

Examples of structs are strings, floats, and ints.

If we declare an int variable and assign it some value and then give it to a method call, any changes made to that parameter within the method won’t affect our variable, for example:

```
void ChangeNumber(int number)
{
    number += 5;
}

int someNumber = 5;

ChangeNumber(someNumber);

Debug.Log(someNumber); //Still logs 5.
```

This is because `int` is a value type. It's a struct, not a class. When you give your "someNumber" variable as a parameter, you're really creating a copy of it and passing that into the method call, so that any changes the method makes to the parameter are not reflected on our variable. They don't point to the same data.

A parameter marked with "out" is a way of navigating around this. `RaycastHit` is a struct, and thus, it's passed as a value type. But the raycast method declares the `RaycastHit` parameter as "out", and thus, any changes made to it in the method will affect the same data we pass in. Our variable and the parameter will both point to the same instance, as if it were a class. If the raycast method detects a hit, it can then fill up the `RaycastHit` with information that we can access through our variable.

That being said, the parameters we give to our `Physics.Raycast` call are in the order of

- The Ray to cast
- The "out" `RaycastHit` to fill with data about the hit, if one occurs
- The maximum distance to cast the ray, which we give `Mathf.Infinity`
- The value of the layer mask to use

When you give a layer mask to a method like this, it usually doesn't ask you for the `LayerMask` instance itself. It asks for an `int` value. To get this, just reference the ".value" member of the `LayerMask`, and it will work its magic behind the scenes.

You can call `Physics.Raycast` with many different kinds of parameters – it has 13 overloads! You can leave out the "hit", you can substitute a `Ray` for its individual components (origin and direction) and other such odd cases, but the one we're using gives us all the control we need.

The raycast method is wrapped in an "if" because it returns true if it hits something and false if it does not. Using this, we can react accordingly: if the ray didn't hit the stage, we deactivate the highlighter and set "cursorIsOverStage" to false. If it did hit the stage, we use the only member of "hit" we need to concern ourselves with: ".point". This is the position that the ray struck on the collider it hit (the stage).

We use some math to round the number out to a multiple of 10. The equation isn't terribly complex: we multiply the value by .1f first, so that, for example, a value of 14 becomes 1.4 instead. Then we round that value, moving to the nearest multiple of 1 – so 1.4 would become 1 instead. After that, we can just multiply by 10 to scale it back up. Thus, 14 goes to 1.4, then to 1, and then back to 10. We do this for both axes, but the Z axis then gets clamped between -80 and 80. This prevents the user from placing towers too close to the bottom or top of the stage.

We set the Y value to .2, which is half of the height of our Highlighter cube. That keeps it nice and flush against the stage plane.

After performing our math and setting the position up, we certainly wouldn't want to forget to actually apply it to the highlighter. We also make sure the highlighter is active and `cursorIsOverStage` is set to true.

This gives us a small portion of the functionality we need. There's still much to do to make our build mode work like we want it to!

The Dictionary

To store towers that have been built, we're going to use a new type of collection – not a List or an array, but a Dictionary. It's a very useful type of collection that stores items not by an index, but by a “key.” The Dictionary is a generic class, taking two generic types when declared: the key type and the value type.

The keys are what identify the values (items) stored within the Dictionary. Whenever you want to store an item in a Dictionary or get an item from a Dictionary, you always supply a key. Rather than giving an int index like in arrays and Lists, we give an instance of the key type and get back the associated value. If there is no value stored in the Dictionary by that key, we get an error.

Shortly put, the key is how we identify the value. Every item in the Dictionary is paired with a key, and there's only one value per key.

In our case, the key is a `Vector3` for the tower position, and the value is the Tower itself. Declare this Dictionary beneath your other private variables:

```
private Dictionary<Vector3, Tower> towers = new Dictionary<Vector3,
Tower>();
```

The first type, `Vector3`, is the key. The second type, `Tower`, is the value. This means we store our Tower instances by their position on the stage. Whenever we add a tower, its key is simply the `Tower.transform.position`. This works perfectly for us, because we know that there will only ever be one Tower in the same position at a time – we don't plan on allowing users to build towers inside of each other, after all. It also means that all we need to do to grab a Tower instance at a location is to pass in the location as a key to our Dictionary. If a tower exists there, we'll get it. And since we're neatly rounding the position of our highlighter to multiples of 10, we can use its position to get the Tower at its location (if there is one).

Getting and setting from a Dictionary is done with indexing, just like arrays and Lists, using the [] syntax. Instead of putting an index in, we put a key.

This means that, if we want to get the Tower at our highlighter position, we can simply type

```
towers[highlighter.position]
```

To set a Tower when we build it, we can do

```
towers[somePosition] = someTower;
```

Most importantly, we need to know if a tower exists at a location before we allow the user to build there. To do this, the Dictionary provides us with a method “ContainsKey”. This method takes a key and returns true if there is a value associated with it or false if there is not.

To check if a tower exists at “somePosition”, we need only do this:

```
//Check if the tower exists:
if (towers.ContainsKey(somePosition))
{
    //If it does exist, we can safely grab it:
    var tower = towers[somePosition];
}
```

But enough theory – let’s just use our Dictionary to see it in action. We’ll implement the OnStageClicked method, which we already set to call whenever the user left-clicks while cursorIsOverStage.

In this method, we behave based on whether or not a build button is currently selected.

If a build button is selected, clicking the stage should attempt to build the tower, assuming we have enough money to do so.

If no build button is selected, then clicking the stage will select the tower under the cursor, if there is one. Later, we’ll position the Tower Selling Panel over the selected tower, but for now, we just need to set “selectedTower”, make sure the Tower Selling Panel is active (and thus visible), and reset the refund text to read the correct amount of gold:

```
void OnStageClicked()
{
    //If a build button is selected:
```

```

if (towerPrefabToBuild != null)
{
    //If there is no tower in that slot and we have enough gold to
    build the selected tower:
    if (!towers.ContainsKey(highlighter.position) && gold >=
        towerPrefabToBuild.goldCost)
    {
        BuildTower(towerPrefabToBuild,highlighter.position);
    }
}
//If no build button is selected:
else
{
    //Check if a tower is at the current highlighter position:
    if (towers.ContainsKey(highlighter.position))
    {
        //Set the selected tower to this one:
        selectedTower = towers[highlighter.position];

        //Update the refund text:
        sellRefundText.text = "for " + Mathf.CeilToInt(selectedTower.
            goldCost * selectedTower.refundFactor) + " gold";

        //Make sure the sell tower UI panel is active so it shows:
        towerSellingPanel.gameObject.SetActive(true);
    }
}
}

```

The `BuildTower` method is a new one, which we'll declare in a second. It takes the Tower prefab to build and the position to build it at.

Since we don't work with fractions of gold, we call `Mathf.CeilToInt` when we calculate the refund amount. The term "Ceil" means "round the fraction up." If there's even a tiny fraction, it gets rounded up – for example, 2.004f becomes 3. "ToInt" implies that we want the float to get converted to an int when it's returned. There is an alternative that's just "`Mathf.Ceil`" which returns the value back as a float instead.

Let's declare BuildTower before we forget:

```
void BuildTower(Tower prefab,Vector3 position)
{
    //Instantiate the tower at the given location and place it in the
    Dictionary:
    towers[position] = Instantiate<Tower>(prefab,position,Quaternion.
    identity);

    //Decrease player gold:
    gold -= towerPrefabToBuild.goldCost;

    //Update the path through the maze:
    UpdateEnemyPath();
}
```

Here we see the assignment of a value in the “towers” Dictionary. We use the position as a key and assign a Tower reference as the value, for which we Instantiate a new prefab instance. We subtract the gold cost from our current gold and then call a method “UpdateEnemyPath”. We'll declare that method later and implement it in the next chapter. It's how we'll find a path through the maze for our enemies to take.

OnClick Event Methods

With that in place, we still have no way of building a tower. We need to declare methods to call when our build buttons are clicked, and then we must hook those methods up to the OnClick event.

As we established before, any public method with a single parameter of a serializable type can be called through a UI event. We have two things to do when a build button is clicked: set the selected button Image component and set the towerPrefabToBuild. Since we can't use two parameters for a single method call, we'll have to split the functionality into two separate methods:

```
public void OnBuildButtonClicked(Tower associatedTower)
{
    //Set the prefab to build:
    towerPrefabToBuild = associatedTower;
```



```

        //Clear selected tower (if any):
        DeselectTower();
    }

    public void SetSelectedBuildButton(Image clickedButtonImage)
    {
        //Keep a reference to the Button that was clicked:
        selectedBuildButtonImage = clickedButtonImage;

        //Set the color of the clicked button:
        clickedButtonImage.color = selectedBuildButtonColor;
    }

```

To ensure that the Tower Selling Panel disappears and the selected tower clears when we click a build button, we run `DeselectTower()` in the first method. We also store the `Image` component of the clicked build button. This `Image` component has its own “color” field that works in conjunction with the various color fields present in the `Button` component. By default, the `Image` color is white, which doesn’t affect the color of the button. Whatever the `Button` defines as the color is what it will be. But by setting this color, we can mix the `Image` color with the `Button` color instead, giving us an easy way to replace the color without having to set all of the various color fields of the `Button` component. This way, we don’t have to remember what to set the colors back to when we want to revert the color to its normal state. We just set the `Image` color back to white, and the `Button` colors take over.

Notice that we’ve made the methods public. We won’t be able to assign them in the events if they’re private. To be able to access our methods, we need to compile the code so the methods are detected. Since we’re trying to call methods that we haven’t declared yet, we’ll get compiler errors when we save and head over to the editor again. To combat that, we’ll have to declare everything we haven’t declared yet. We’ll declare them as simple one-line methods for now, with empty curly braces – just to appease the compiler until we’re ready to code the contents of the methods:

```

void PositionSellPanel() {}

void UpdateCurrentGold() {}

public void DeselectTower() {}

void DeselectBuildButton() {}

void UpdateEnemyPath() {}

```

DeselectTower will be called as an event when the user clicks the “X” in the Tower Selling Panel, so it’s important that we declare it as public. The rest are all private.

With that, we should have no errors left to trudge through when we save and return to the editor.

For now, all we’ll be setting up is our Arrow Tower build button, since we don’t have the other towers or their prefabs set up yet. Select the Arrow Tower build button and head to the bottom of the Button component in the Inspector:

- Where the `OnClick` event is listed, click the little checkmark in the bottom right to add an event.
- Beneath the dropdown field with “Runtime” selected, there should be an Object field set to None. Drag the Player Camera GameObject from the Hierarchy onto this field.
- The field titled “No Function” will become available. Clicking it, you’ll see options for `GameObject`, `Transform`, `Camera`, and more.
- Click the Player option. More options will unfold, each one a variable or method that we can interact with. Select our method “`OnBuildButtonClicked`.”
- A field will pop up for the Tower prefab associated with the button. This is our parameter. Drag the Arrow Tower prefab from the Project onto this field.
- Again, click the Plus, drag the Player Camera GameObject onto the first field, and, this time, select Player ► `SetSelectedBuildButton`. Scroll up in the Inspector and drag the Image component from this build button, dropping it onto the parameter field.

When you’re done, the `OnClick` event should look like Figure 29-8.

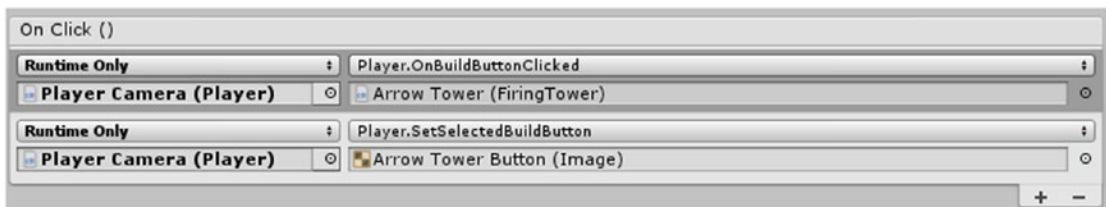


Figure 29-8. The `OnClick` event with our two method calls set up to occur

At last, a portion of the functionality is operational. Playing now, you should be able to click the Arrow Tower button to make it turn green, indicating that it is selected. Pressing Escape won't do anything yet, because we haven't implemented `DeselectBuildButton` yet, but clicking the stage should build an instance of the tower.

Let's implement that method. We declared it already to get the compiler off our back, but we never filled it with code:

```
void DeselectBuildButton()
{
    //Null the tower prefab to build, if there is one:
    towerPrefabToBuild = null;

    //Reset the color of the selected build button, if there is one:
    if (selectedBuildButtonImage != null)
    {
        selectedBuildButtonImage.color = Color.white;
        selectedBuildButtonImage = null;
    }
}
```

It's simple enough. It's already being called when we press Escape, through our `BuildModeLogic` method. It nulls out the variables associated with the currently selected build button and reverts the Image component back to the default white color so the button doesn't appear selected anymore.

Now let's implement `UpdateCurrentGold`:

```
void UpdateCurrentGold()
{
    //If the gold has changed since last frame:
    if (gold != goldLastFrame)
        //Update the text to match:
        currentGoldText.text = gold + " gold";

    //Keep track of the gold value each frame:
    goldLastFrame = gold;
}
```

With that in place, building a tower should immediately cause our gold to decrease in the bottom-left corner.

We still can't sell towers, though. We need to position the sell panel constantly to keep it above the selected tower by implementing the `PositionSellPanel` method:

```
void PositionSellPanel()
{
    //If there is a selected tower:
    if (selectedTower != null)
    {
        //Convert tower world position, moved forward by 8 units, to screen
        space:
        var screenPosition = Camera.main.WorldToScreenPoint(selectedTower.
            transform.position + Vector3.forward * 8);

        //Apply the position to the tower selling panel:
        towerSellingPanel.position = screenPosition;
    }
}
```

Here, we use a new Camera method: `WorldToScreenPoint`. It takes a position in world space and converts it to a point on the screen. The position we give is the selected tower position, but we move it forward by 8 units to place the panel somewhere “above” the Tower on our screen.

We already set the selected tower when the stage is clicked while a tower is not being built. With this in place, you should be able to play the game, click the Arrow Tower build button, click the stage to build one, then press Escape to deselect the build button, and then click the tower you built to select it. The panel should show above it, showing the correct amount of gold refund in the text, shown in Figure 29-9.



Figure 29-9. *The Tower Selling Panel showing above an Arrow Tower*

Since we haven't set up the Sell or X button to do anything yet, clicking them will be somewhat disappointing. Let's change that.

We need to implement the methods we plan on calling with our events. First off is a method that occurs when the Sell button is clicked:

```
public void OnSellTowerButtonClicked()
{
    //If there is a selected tower,
    if (selectedTower != null)
        //Sell it:
        SellTower(selectedTower);
}
```

And, of course, we must implement that SellTower method:

```
void SellTower(Tower tower)
{
    //Since it's not going to exist in a bit, deselect the tower:
    DeselectTower();

    //Refund the player:
    gold += Mathf.CeilToInt(tower.goldCost * tower.refundFactor);

    //Remove the tower from the dictionary using its position:
    towers.Remove(tower.transform.position);
}
```

```

    //Destroy the tower GameObject:
    Destroy(tower.gameObject);

    //Refresh pathfinding:
    UpdateEnemyPath();
}

```

Again, we Ceil the refund value and add the money back to the player's current gold. To remove the Tower from our "towers" Dictionary, we call the Remove method. It takes the key of the value we want to remove. Since the key is the position of the associated tower, all we need to do is reach into the given tower parameter and reference its "transform.position".

Then we destroy the tower GameObject.

Since a Tower just got removed from the stage, we need to update the enemy path again to ensure that it finds the easiest path through the stage, considering a better route may have just opened up with the loss of that tower. That's why we call UpdateEnemyPath.

We also need to code the contents of our method to deselect a tower:

```

public void DeselectTower()
{
    //Null selected tower and hide the sell tower panel:
    selectedTower = null;
    towerSellingPanel.gameObject.SetActive(false);
}

```

You know how to add events now. It's the same thing we did with the build button.

For the Sell button, add an event that calls Player ► OnSellTowerButtonClicked.

For the X button, add an event that calls Player ► DeselectTower.

Neither of these methods has a parameter to reference, so we just need to point at the method and be on our way.

With that, the core functionality of Build Mode will be operational:

- Clicking a Build Button will select it, turning it green. For now, the other three build buttons aren't wired up with their events because we haven't made the tower yet, so they won't do anything, but the Arrow Tower will.

- Clicking the stage with a build button selected will build the tower there, if the player has the money to do so.
- Pressing Escape with a build button selected or a tower selected will clear the selection.
- Clicking a tower without a build button selected will select the tower, showing the sell panel above it. Clicking Sell will sell the tower and refund us, while clicking X will deselect the tower so the panel stops showing.
- Clicking the build button while a tower is selected will also clear the tower selection.
- Any change in gold from buying or selling a tower will update the gold indicator in the bottom left.

In the next chapter, we'll start working on our enemy pathfinding and spawning for Play Mode.

Summary

This was a significant chapter with many new concepts to explore. We learned how to

- Set up a Canvas and add UI elements as child GameObjects. The drawing order follows the order of element GameObjects from top to bottom in the Hierarchy: lower elements render on top of higher elements.
- Attach actions that occur on certain events in a UI element. This gives us the ability to call a method from a script or component or to set the value of a property in a component or GameObject. If you want to call a method through an action, the method must be public and must have no more than one parameter.
- Perform a raycast to test for a collision from a starting point, moving along a direction for a maximum amount of distance.

- Convert to a screen position from a world position using a Camera.
- Use a LayerMask to define the layers a raycast can collide with.
- Get a Ray that shoots out of the Camera from a given screen position within the Camera.
- Use a Dictionary to store objects by key-value pairs.

CHAPTER 30

Play Mode

With our Build Mode giving the player a means of buying and selling towers, we can now begin to work on Play Mode. During Play Mode, enemies spawn at the spawn point near the top of the stage and navigate around towers using pathfinding, traveling toward the leak point at the bottom of the stage.

In this chapter, we'll implement pathfinding whenever the player buys or sells a tower, keeping an updated path from spawn point to leak point. If the path is blocked, we'll prevent the player from clicking the Play button until the path becomes unblocked (they'll have to sell towers to clear the way).

We'll implement both the `GroundEnemy` and `FlyingEnemy` types, which inherit from our base `Enemy` script but provide their own implementations of movement. `GroundEnemy` follows the path, and `FlyingEnemy` just takes a straight shot, going over the towers.

We'll also implement a system for spawning the enemies at specific intervals so they travel in a single file line and increasing their health each level to keep up with the player building more towers. We'll make it so enemies "leak" when they reach the leak point, taking a life from the player.

Finally, we'll reward the player with gold when the level is finished, or if they've lost all their lives already, we'll end their session by throwing a panel over the UI that displays a "game over" message.

Spawn and Leak Points

Let's set up the spawn point and leak point at each side of the Stage plane. Our enemies will be 5 units wide and long, so we'll make "plates" on the floor that are the same size:

- Create an empty `GameObject`. Name it `Spawn Point`. Set its position to (0, 0, 96).
- Add a child `Cube`. Set its position to (0, .1, 0) and its size to (5, .2, 5).

- Copy-paste the Spawn Point and rename it to Leak Point. Make sure you've selected the root itself, not the Cube inside it. Set its position to (0, 0, -96) to put it at the bottom side of the stage instead of the top.
- I'll create Spawn Point and Leak Point materials and apply them to their corresponding cube, giving the spawn point a cyan color with a hex value of 92F09A and a red color with a hex value of FF2227 for the leak point.

Once you're finished, the stage should look like Figure 30-1 when it's viewed from above: just a long, rectangular strip with a square plate on each end.

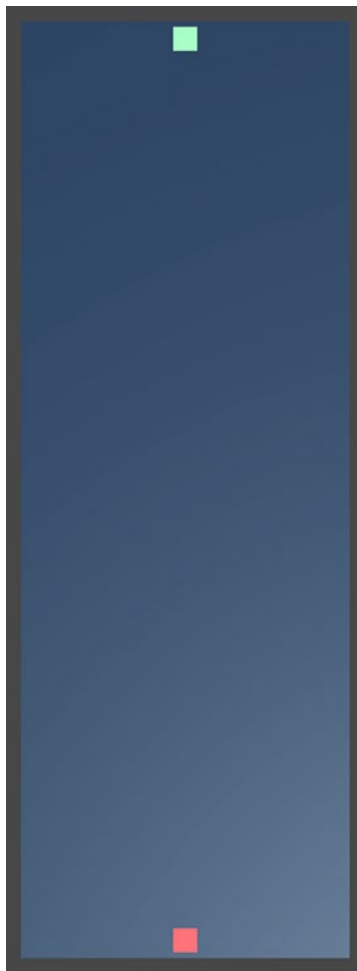


Figure 30-1. The Stage plane viewed from above, with the Spawn Point at the top and the Leak Point at the bottom

In your Player script, add two reference variables pointing to the Transform of the spawn point and leak point:

```
public Transform spawnPoint;
public Transform leakPoint;
```

Save and return to the editor to set these references so they're ready to go when we need them.

Locking the Play Button

When we implement pathfinding, we'll be "locking" the Play button if the way from the spawn point to the leak point is blocked after the player builds a tower. We want a means of preventing the player from clicking that button, and we want to put some text above the button explaining the situation.

To do this, we'll position a Panel over the Play Button, rendering on top of it – meaning the panel must be lower in the Hierarchy than the Play Button. While this panel is active, it will block the Play Button. Clicking the button won't work because the click will be "stolen" by the panel, which doesn't do anything when we click it.

Once the panel is in place, we'll reference it in the Player script, from which we can activate it if the path is blocked and deactivate it once it's unblocked. The text message associated with the panel will be a child of it so that it also gets activated and deactivated.

Let's create the panel:

Right-click the Canvas and add a Panel. Name it Play Button Lock Panel. It should already be below the Play Button in the Hierarchy, making it display over the button:

- Set the anchor preset to Bottom Center.
- Size the panel to 240 width and 70 height.
- Center the panel over the Play Button, or just hold Alt and apply the anchor preset again.
- Change the color field of the panel Image component to a red with a hex value of FD5757. Change the alpha to .7, making it partially transparent.

The panel will act as something of a screen that lays out over the button, adding a washed-out red color to it. Figure 30-2 compares the Play Button with and without the panel showing over it.



Figure 30-2. *The Play Button in its normal state (top) vs. when the Lock Panel is active (bottom)*

Now let's add the text that will show above the Play Button while the panel is active:

- Right-click the Play Button Lock Panel and add a child UI ► Text. Change its size to 340 width by 80 height. Set its Y position to 85.
- Make the text bold with a font size of 22.
- Type this message into the text box in the Text component: *"Towers are blocking enemies from reaching the leak point! Can't play until a path is cleared!"*
- Set the text color to a pale red with a hex value of FF4949.

When you're finished, the button should look like Figure 30-3 while the panel is active. By default, make the panel inactive so it doesn't show when the game first starts.

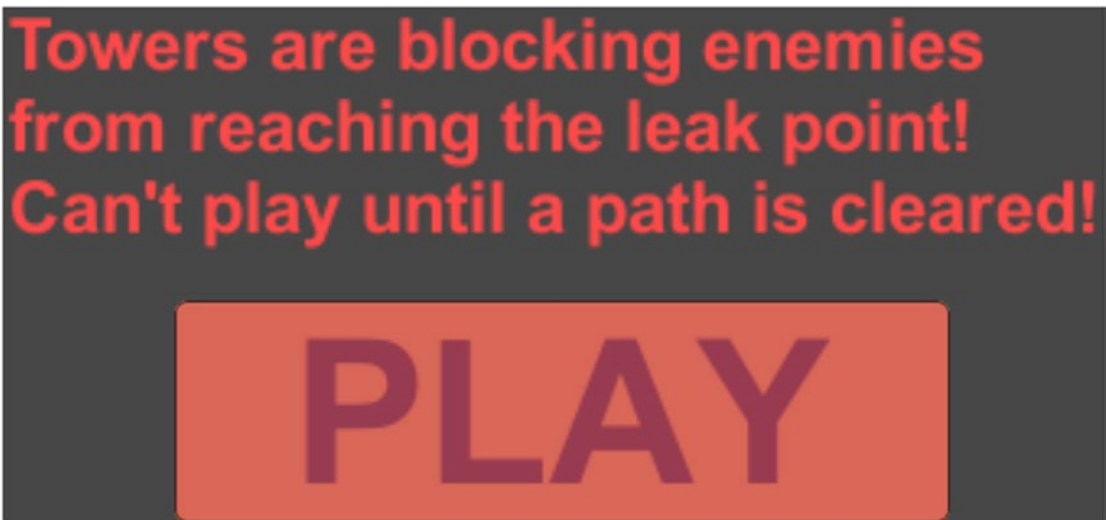


Figure 30-3. *The Play Button while the Play Button Lock Panel is active*

We'll be needing a reference pointing to the GameObject of the panel so we can activate and deactivate it.

Add this reference to the Player script:

```
[Tooltip("Reference to the sell button lock panel GameObject.")]
public GameObject sellButtonLockPanel;
```

And of course, don't forget to return to the editor and set the reference.

Pathfinding Setup

Unity provides options for pathfinding and associated AI, like local avoidance (AI that prevents navigating objects from bumping into each other while moving toward their goal). It even has solutions for dealing with slopes and open space that can be jumped over. We don't need this sort of functionality for our game, though. We just want to find a path around our towers on our flat stage, no jumping or slopes involved, and we want to move our enemies with our own scripts. Luckily, the fancy features don't get in our way or force us to use them. We'll be able to call a method that calculates a path from point A to point B and gives us the resulting points along the path in sequence.

There are some steps we'll need to take to set up pathfinding for our project. Firstly, we need to mark the Stage as Navigation Static. This is done in the Navigation window, which you can open through Window ► AI ► Navigation on the top bar. Click the Object tab at the top of the Navigation window and select the Stage in the Hierarchy window. This should change the Navigation window to show a checkbox field titled "Navigation Static" for the Stage, shown in Figure 30-4. Check that box.

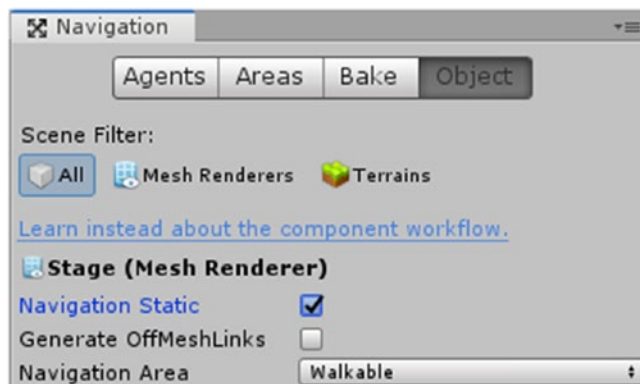


Figure 30-4. Navigation window while selecting the Stage. The Navigation Static checkbox has been ticked

This marks the Stage as a static part of the world that we can find a path over. To update the navigation data after making a change like this, we must go to the Bake tab in the Navigation field, shown in Figure 30-5.

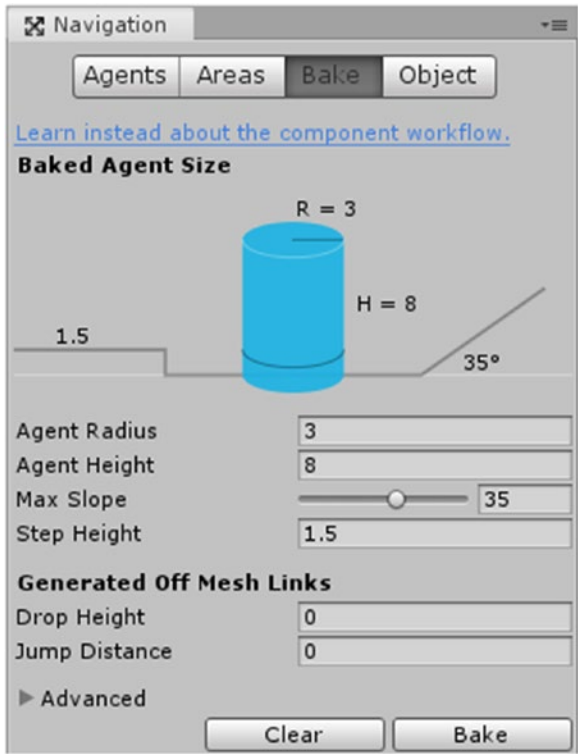


Figure 30-5. *Bake tab of the Navigation window*

This tab displays information about the size of the “baked agent.” An agent is the term for an object that’s using the navigation. They are resembled as cylinders:

- **Agent Radius** is the distance from the center of the cylinder to the edge. Since our enemies are 5 units wide and long, a radius of 2.5 would make the agents the same size as the enemies. We’ll set it instead to a value of 3 to give a little extra space between the enemies and the towers.
- **Agent Height** is the number of units tall the cylinder is. Our enemies are 8 units tall.

- **Max Slope** is the highest angle of a slope the agent will be able to walk up.
- **Step Height** is the maximum height an agent can step up or down.

The latter three values don't really mean anything to us, since we have such a basic use case.

Once you've set the values, click Bake. Unity will generate a NavMesh asset resembling the navigation data associated with the scene. This will be located in the Project in a folder named after the scene.

The baked NavMesh will show in the Scene view as a light-blue overlay on the Stage as long as the Navigation window is around. This shows us the walkable areas for our agents. The center of the agent can walk anywhere in the highlighted space. Since it's the center of the agent, it will be spaced away from the edges by an amount equal to the agent radius field. This ensures that the agent will only be allowed to go into places which won't cause it to stretch outside the walkable area.

We still haven't defined our towers as obstacles that block navigation. The navigation system doesn't automatically qualify any collider as an obstacle. We have to add a NavMeshObstacle component to our towers to specify them as an obstacle that should be navigated around:

- Open the Arrow Tower prefab by double-clicking it in the Project window and add a NavMeshObstacle component to its root GameObject.
- Set the center to (0, 3, 0) and the size to (10, 6, 10).
- Check the "Carve" field. This field resembles whether or not the NavMeshObstacle should carve out a space from the NavMesh. If this was set to false, the obstacle would not affect the NavMesh itself, but would be avoided by any agents using Unity's built-in NavMeshAgent component. We aren't using the component – we're just finding a path on the NavMesh. This means the obstacle won't do anything for us unless we enable the Carve feature.
- Save the prefab and return to the scene.

We'll do this for all of our other tower types as well; otherwise, enemies will navigate right through them.

With that in place, you can play the game, build some arrow towers, and then switch to the Scene view while the Navigation window is open somewhere in your editor. You should see the NavMeshObstacle components doing their job, carving out a space in the NavMesh, as shown in Figure 30-6.

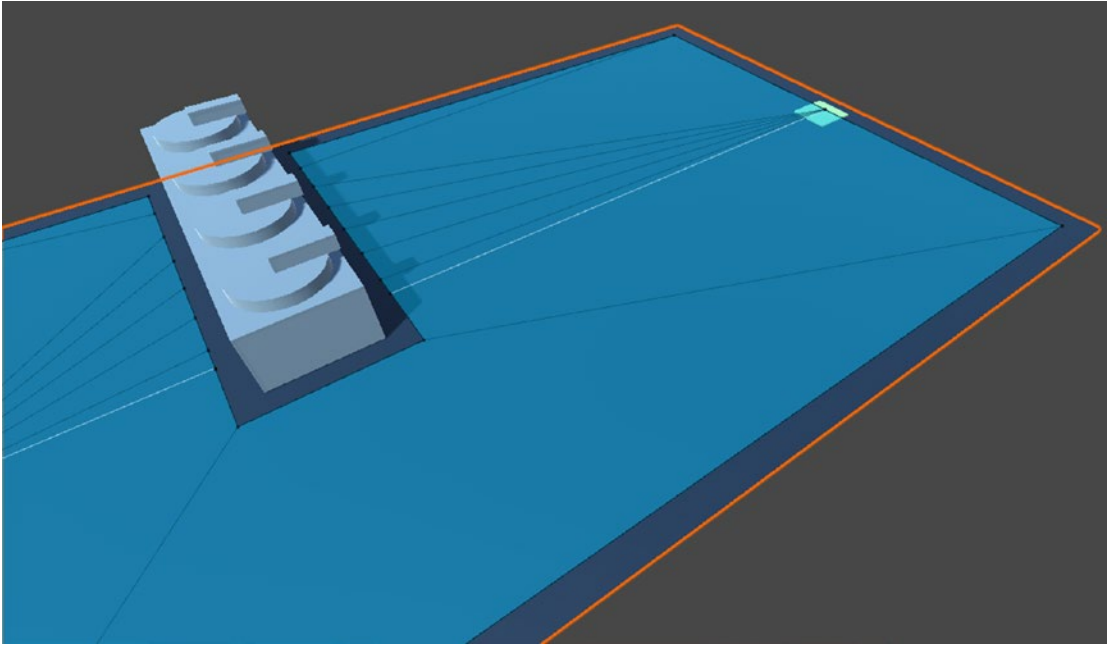


Figure 30-6. Some arrow towers are placed on the Stage, causing the walkable area of the NavMesh to wrap around them

Finding a Path

In the previous chapter, we declared an empty method called `UpdateEnemyPath` and ran it whenever a tower was built or sold. This is where we'll do our pathfinding.

Because the path relates to ground enemies, we're going to declare the path as a static variable in the `GroundEnemy` script. You'll recall from Chapter 9 that a static variable is a variable that is tied not to each instance of the class, but to the class type itself. With static variables, one instance of the variable exists for the whole class, and any instance of the class which uses this variable will be pointing to the same thing.

A common example of how one might use this is to create a count of the total number of instances of a class that have been created:

```
public class ExampleScript : MonoBehaviour
{
    public static int instancesCreated = 0;

    void Start()
    {
        instancesCreated += 1;
    }
}
```

The variable “instancesCreated” is given the “static” keyword. In the Start method, we add 1 to that variable. This will make each script instance add 1 to the total count, and since the variable is static, they’re all pointing to the same thing.

Since they are tied to the class type itself, not its individual instances, static variables can be accessed by reaching into the class name. For example, a different script could grab the instancesCreated at any moment by referencing “ExampleScript.instancesCreated”.

Let’s put this to practice. Create a GroundEnemy script. Open it and make it inherit from our base Enemy class, so the class declaration line reads like this:

```
public class GroundEnemy : Enemy
```

To access necessary navigation-related methods, we’ll need to write this “using” **at the top of the script file**:

```
using UnityEngine.AI;
```

This gives us access to the NavMeshPath class, which is what we will use to store a path given to us by the navigation system. Declare this variable inside the class script:

```
public static NavMeshPath path;
```

This is our static variable pointing to the path that our ground enemies will take. Since they’ll all take the same path anyway, it wouldn’t make sense to give them an instanced (non-static) variable that has to be set for each ground enemy when it is Instantiated. With our static variable, we can set the path up in our Player script, and our GroundEnemy instances will have access to it at all times to begin moving along the path as soon as they spawn.

The NavMeshPath needs to be initialized to an instance by calling the constructor, but we can't do it right where it is in the script class declaration. It must be done from Start. We'll do it from the Player script Start method, since we won't have any GroundEnemy instances starting up before the path is needed – for example, the player will be building towers and updating the path before the first enemy ever spawns.

We'll also need to update the path once at the start, before any enemies spawn, to ensure that if the player starts the level without building any towers, the path will still be set up to go from the spawn point to the leak point.

Add these two lines **in the Player Start method**:

```
GroundEnemy.path = new NavMeshPath();
UpdateEnemyPath();
```

Since the variable is static, we access it through the class name from anywhere we like. We assign a new instance of the NavMeshPath here, just once at the start of the game. If we neglect this step, we'll get a compiler error when we try to find the path because the NavMeshPath can't be null.

Now **add the “using UnityEngine.AI;” line to the Player script** as well so we can access the NavMesh class. This class has the method “CalculatePath”, which is a simple means of pathfinding from one point to another and storing the path in a given NavMeshPath instance.

We need to call this method from the UpdateEnemyPath method that we declared earlier. However, if we simply call the method there, we'll run into a problem. Even though we always call UpdateEnemyPath after we Instantiate a newly bought tower or Destroy a sold tower, the effects of creating and destroying the towers don't always happen immediately. It might take until next frame for our NavMeshObstacle to start or stop affecting the NavMesh. Thus, we can't just run the CalculatePath method immediately, because it might still calculate the path before the obstacle is added or removed. We need to do it shortly after the UpdateEnemyPath method is called.

To accomplish this, we'll declare a PerformPathfinding method:

```
void PerformPathfinding()
{
    //Pathfind from spawn point to leak point, storing the result in
    GroundEnemy.path:
    NavMesh.CalculatePath(spawnPoint.position, leakPoint.position,
        NavMesh.AllAreas, GroundEnemy.path);
```

```

if (GroundEnemy.path.status == NavMeshPathStatus.PathComplete)
{
    //If the path was successfully found, make sure the lock panel is
    inactive:
    sellButtonLockPanel.SetActive(false);
}
else //If the path is blocked,
{
    //Activate the lock panel:
    sellButtonLockPanel.SetActive(true);
}
}

```

And then Invoke that method to occur in a brief moment in the UpdateEnemyPath method:

```

void UpdateEnemyPath()
{
    Invoke("PerformPathfinding",.1f);
}

```

We call NavMesh.CalculatePath, as mentioned before. The first parameter is the start point of the search, and the second parameter is the end point of the search. We use the position of our spawn point and leak point. The third parameter is asking for an area mask. This is like a layer mask, but it corresponds to area types that we can set up in the Navigation window. These types allow you to specify different types of ground, each one with its own “cost” value. The pathfinding operation will account for the cost of areas when trying to find the ideal path. This can be used to define areas of ground that are walkable, but more costly to walk on than others – for example, ground that burns those who walk on it would have a high cost, and thus, the pathfinding might choose to go around it rather than cutting through it. We don’t have a need for this feature, so we just reference the static member NavMesh.AllAreas area mask to fill this parameter. Lastly, we give GroundEnemy.path as the path to fill with data.

Afterward, we can reference the path “.status” member, which is an enum “NavMeshPathStatus” that will depict whether or not the path made it to the destination. If it did, the path will be PathComplete. If the path could not make it all the way to the end point, it will be PathPartial.

We activate or deactivate the sell button lock panel based on whether or not the path was complete.

At that, you should be able to play the game and see the effects. If you build a line of towers blocking the path, the lock panel will activate, preventing you from pressing Play, as depicted in Figure 30-7.

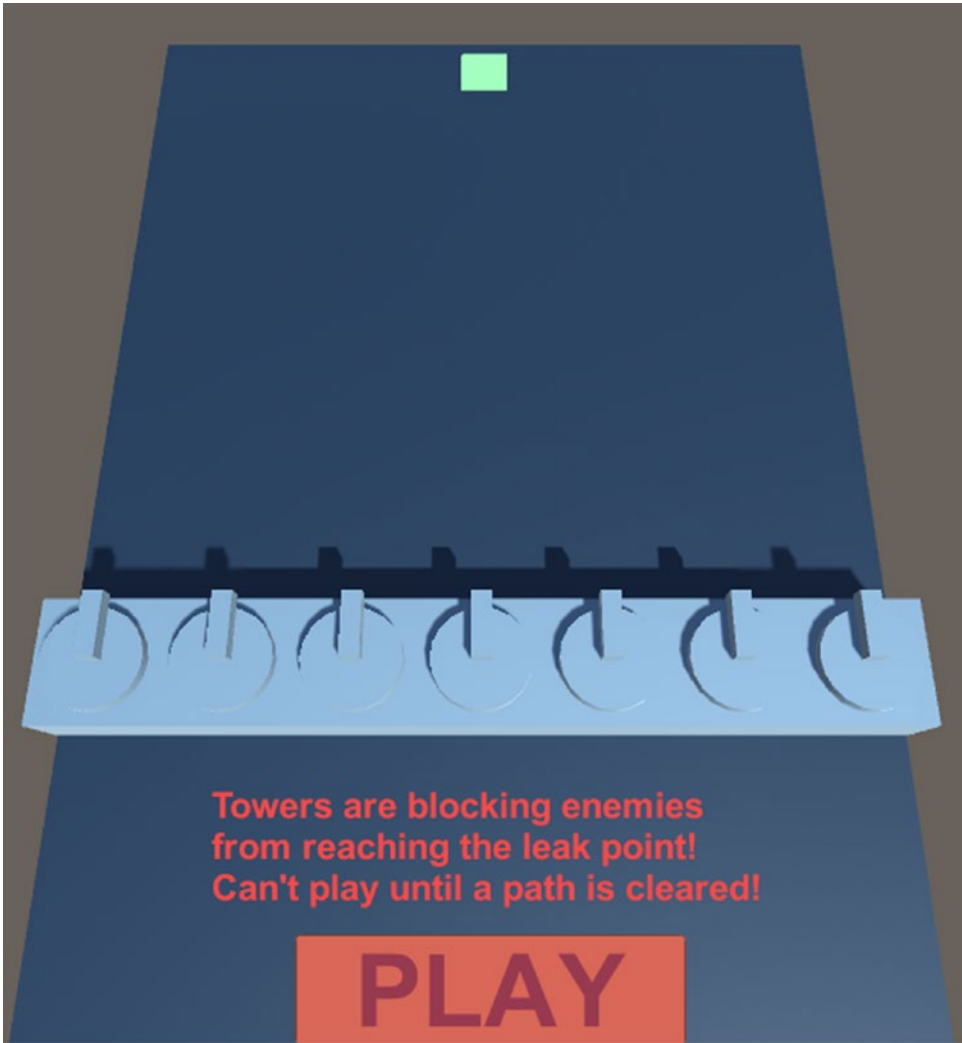


Figure 30-7. *The Play button has locked because we have blocked the way with our towers*

Selling one of these towers to unblock the way will cause the button to unlock.

Play Mode Setup

The Play button will start the level when it is pressed and begin spawning enemies. We'll spawn one enemy at a time until a certain number of enemies have been spawned. Enemies will either die to towers or leak by reaching the leak point. Either way, they get destroyed. Once all enemies have finished spawning and no enemies are left in the game, the level is complete. If the player has no health left, they've lost, so we'll toggle on a panel that covers the screen, telling them how disappointing they are. If not, we just return to build mode, give the player money, and increase the current level by 1.

Let's declare the relevant variables for all of this logic at once in the Player script. We'll make a new heading, down beneath the Build Mode heading, and add these variables:

```
//Play Mode:
[Header("Play Mode")]
[Tooltip("Reference to the Build Button Panel to deactivate it when play
mode starts.")]
public GameObject buildButtonPanel;

[Tooltip("Reference to the Game Lost Panel.")]
public GameObject gameLostPanel;

[Tooltip("Reference to the Text component for the info text in the Game
Lost Panel.")]
public Text gameLostPanelInfoText;

[Tooltip("Reference to the Play Button GameObject to deactivate it in play
mode.")]
public GameObject playButton;

[Tooltip("Reference to the Enemy Holder Transform.")]
public Transform enemyHolder;

[Tooltip("Reference to the ground enemy prefab.")]
public Enemy groundEnemyPrefab;

[Tooltip("Reference to the flying enemy prefab.")]
public Enemy flyingEnemyPrefab;

[Tooltip("Time in seconds between each enemy spawning.")]
public float enemySpawnRate = .35f;
```

```
[Tooltip("Determines how often flying enemy levels occur. For example if
this is set to 4, every 4th level is a flying level.")]
public int flyingLevelInterval = 4;

[Tooltip("Number of enemies spawned each level.")]
public int enemiesPerLevel = 15;

[Tooltip("Gold given to the player at the end of each level.")]
public int goldRewardPerLevel = 12;

//The current level.
public static int level = 1;

//Number of enemies spawned so far for this level.
private int enemiesSpawnedThisLevel = 0;

//Player's number of remaining lives; once it hits 0, the game is over:
public static int remainingLives = 40;
```

The first chunk of variables are all references to various things, which we'll need to set up now:

- **buildButtonPanel** should be set to the Build Button Panel in our Canvas so we can deactivate and activate it accordingly when entering play mode or build mode.
- **gameLostPanel** will be a UI panel we'll create that lays over the screen when the player has lost all their lives. We'll create it in a bit.
- **gameLostPanelInfoText** is a text element inside the Game Lost Panel that tells us how many lives we had left (so we know how badly we lost) and what level we lost at. We'll have to set the text to make it have the proper information.
- **playButton** points to the Play Button. Like the buildButtonPanel, we'll be deactivating it in play mode to hide it and reactivating it when build mode begins again.
- **enemyHolder** should be set to a new, empty GameObject named Enemy Holder. We'll use this as the parent to all enemies; and, when all enemies are destroyed, we know the level has ended.

- **groundEnemyPrefab** and **flyingEnemyPrefab** will be set when we create the associated prefabs for enemies.

The remaining variables are explained in their tooltips. We have a means of changing the time between each enemy spawning, how many ground levels must pass before a flying level occurs, how many enemies will spawn in each level, and how much gold the player earns at the end of the level, a static variable for the current level and the lives remaining, and a private variable that we'll use to track how many enemies we've spawned so we know when to stop.

Let's create the Game Lost Panel so it's ready to use when we need it:

- Add an instance of our Panel prefab as a child to the Canvas. Name it Game Lost Panel. It should automatically cover the whole Canvas. If not, open the anchor presets dropdown, hold Alt, and click the bottom-right "Stretch" option.
- Add a child Text element to the Game Lost Panel. It should be centered on the panel. Name it Game Over Text. Give it a width of 340. Make it bold, center and justify it, give it a font size of 48, set the Vertical Overflow field to Overflow, and give it a color with a hex value of FFA800.
- Add another child Text element to the Game Lost Panel. Name it Info Text. Give it a Y position of -182 and size of 340×274. Center it horizontally and align it to the top. Give it 24 font size and a hex value of FFC044. Clear the text out; we'll set it through script.

Once you've finished, deactivate the Game Lost Panel by default and drag and drop a reference to it and the Info Text into your Player variables. Make sure it's the lowest child of the Canvas, since we want it to draw over the top of everything else.

Again, make sure you've set up your Enemy Holder (just an empty GameObject) and set the reference to it, as well as the Build Button Panel.

First off, let's declare play mode logic for our Update method. We'll do much the same thing we did with the BuildModeLogic method. Locate this portion of your Update method:

```
//Run build mode logic if we're in build mode:
if (mode == Mode.Build)
    BuildModeLogic();
```

And add an “else” that calls a `PlayModeLogic` method instead:

```
//Run build mode logic if we're in build mode:
if (mode == Mode.Build)
    BuildModeLogic();
else
    PlayModeLogic();
```

Beneath all of your build mode–related methods in the `Player` script, declare the `PlayModeLogic`:

```
public void PlayModeLogic()
{
    //If no enemies are left and all enemies have already spawned
    if (enemyHolder.childCount == 0 && enemiesSpawnedThisLevel >=
        enemiesPerLevel)
    {
        //Return to build mode if we haven't lost yet:
        if (remainingLives > 0)
            GoToBuildMode();
        else
        {
            //Update game lost panel text with information:
            gameLostPanelInfoText.text = "You had " + remainingLives + "
            lives by the end and made it to level " + level + ".";

            //Activate the game lost panel:
            gameLostPanel.SetActive(true);
        }
    }
}
```

To detect when the level is finished, we use the `Enemy Holder`, checking its `Transform.childCount` member, which tells us how many children it has. If it has no children and we’ve already spawned all of the enemies we plan on spawning for that level, we count the level as finished.

If we haven’t run out of lives yet, we run `GoToBuildMode` to switch back to build mode. We’ll declare it soon.

If we have run out of lives, the Game Lost Panel will ruin our fun by sprawling across our screen and preventing us from seeing anything or hitting any buttons.

Let's declare a `GoToPlayMode` to define the logic that occurs when it's time to switch from build mode to play mode:

```
void GoToPlayMode()
{
    mode = Mode.Play;

    //Deactivate build button panel and play button:
    buildButtonPanel.SetActive(false);
    playButton.SetActive(false);

    //Deactivate highlighter:
    highlighter.gameObject.SetActive(false);
}
```

It's pretty self-explanatory: we don't want the highlighter or build button panel to show during play mode, we just deactivate them.

Now, let's switch back to build mode from play mode, which we already scripted to occur when the level ends and we still have lives left:

```
void GoToBuildMode()
{
    mode = Mode.Build;

    //Activate build button panel and play button:
    buildButtonPanel.SetActive(true);
    playButton.SetActive(true);

    //Reset enemies spawned:
    enemiesSpawnedThisLevel = 0;

    //Increase level:
    level += 1;
    gold += goldRewardPerLevel;
}
```

We reactivate the build button panel. The highlighter will reactivate itself on its own when we mouse over the stage again, as it always does in build mode, so we needn't worry about it. We also reset and increment the relevant variables.

Now, declare a `StartLevel` method that we'll hook up to our Play button `OnClick` event:

```
public void StartLevel()
{
    //Switch to play mode:
    GoToPlayMode();

    //Repeatedly invoke SpawnEnemy:
    InvokeRepeating("SpawnEnemy", .5f, enemySpawnRate);
}
```

Of course, set it up in the Play button to make sure it gets called when the button is clicked. We've done this in the previous chapter with our build button and tower selling button.

The first line of code is obvious – switch to play mode – but the second line is new to us.

Spawning Enemies

To repeatedly spawn an enemy, we'll use this new form of invoking: the `InvokeRepeating` method. This method will keep invoking a method at a given rate. It takes three parameters:

- The **method name** of the method to invoke, as a string.
- The **initial wait time** in seconds. The first call takes this long to occur after we call `InvokeRepeating`.
- The **interval time** in seconds. After the first call occurs, this is the time between each call thereafter.

`InvokeRepeating` will keep invoking our method until we cancel it by calling the `CancelInvoke` method. We can call `CancelInvoke` with no parameters to cancel all ongoing method invokes for the script, or we can give a string parameter for the name of a single method we want to stop invoking.

Our `SpawnEnemy` method will spawn either the ground enemy prefab or the flying enemy prefab based on the level, using an operator we haven't used yet: the `"%"` symbol, called a modulus operator. This operator takes a number value on either side. The number on the left will continuously be subtracted from by the number on the right until the remainder is less than the number on the right. Then, that remainder is returned.

Let's see it in action and declare the method we're invoking:

```
void SpawnEnemy()
{
    Enemy enemy = null;

    //If this is a flying level
    if (level % flyingLevelInterval == 0)
    {
        enemy = Instantiate(flyingEnemyPrefab,spawnPoint.position +
            (Vector3.up * 18),Quaternion.LookRotation(Vector3.back));
    }
    else //If it's a ground level
    {
        enemy = Instantiate(groundEnemyPrefab,spawnPoint.
            position,Quaternion.LookRotation(Vector3.back));
    }

    //Parent enemy to the enemy holder:
    enemy.trans.SetParent(enemyHolder);

    //Count that we spawned the enemy:
    enemiesSpawnedThisLevel += 1;

    //Stop invoking if we've spawned all enemies:
    if (enemiesSpawnedThisLevel >= enemiesPerLevel)
    {
        CancelInvoke("SpawnEnemy");
    }
}
```

We declare a null variable for the enemy we plan on spawning, and then we spawn either a ground or flying enemy prefab, assigning the new enemy to that variable. As you may recall, the `Instantiate` method takes arguments in the order of prefab to spawn, position to spawn it at, and rotation to spawn it with. Our flying enemies are spawned at the spawn point, plus 18 units in the global up direction. We also make sure to make enemies look at the global back direction, which points them toward the leak point.

Our equation to ask “Is this a flying level?” is “`level % flyingLevelInterval`”. The `flyingLevelInterval` will remain at 4 by default all the time, so let’s observe the behavior as the “level” increases.

For levels 1, 2, and 3, the left-side number in our modulus operator is less than 4, so the remainder is just the same as the left-hand value. The result is 1, 2, and 3. We are not on an air level, because the result is not 0.

Once we hit level 4, the left-side number is now equal to the right-hand value, so the right-hand value is subtracted from it once, leaving a remainder of 0. That means it is a flying level.

Then the level becomes 5, 6, and 7. For these, the right-hand value (4) can still be subtracted only once before the remainder is too low to subtract any more, so it gets returned. Again, we get 1, 2, and 3. Then we hit 8, and the right-hand value is subtracted twice now, leaving 0 again – another flying level.

And so on, the flying levels will come at levels 4, 8, 12, 16, and so on, simply by automatically spawning a different prefab when the level is a multiple of 4 (or whatever the variable is set to).

Aside from that, the method then makes the new enemy a child of the `Enemy Holder`, which is how we’ll track how many enemies are left alive. Then we count +1 spawned enemy and cancel the repeating invoke with the `CancelInvoke` call if we’ve reached the target number of enemies spawned.

Now we need to code the `GroundEnemy` and `FlyingEnemy` to make them function correctly.

Before we do this, let’s change the way they set their health. In the base `Enemy` script, we’ll give them a `healthGainPerLevel` variable, declared under their `maxHealth`:

```
public float healthGainPerLevel;
```

We’ll change the `Start` method to set the max health to the base value it is given in the prefab, plus the health gained per level, which, of course, we must multiply by the current level:

```
protected virtual void Start()
{
    maxHealth = maxHealth + (healthGainPerLevel * (Player.level - 1));
    health = maxHealth;
}
```

We multiply by the level – 1 so that at level 1, the max health is the base value given in the prefab, and only levels won after that will add to enemy health.

While we’re at it, we can also declare a method we’ll be using in a bit: a `Leak` method for the base `Enemy` class that we’ll call from our lower classes when they reach the leak point.

Declare this in the `Enemy` class:

```
public void Leak()
{
    Player.remainingLives -= 1;
    Destroy(gameObject);
}
```

We’ll take away a life from the player, referencing the static variable to do so. We’ll also destroy the enemy.

Enemy Movement

Let’s make our `GroundEnemy` movement, and then we’ll work on our `FlyingEnemy`. We already made our `GroundEnemy` script earlier, but we never implemented an `Update` method to make it move.

First, we’ll need some variables. Below the static path variable we declared earlier, add these too:

```
public float movespeed = 22;

private int currentCornerIndex = 0;
private Vector3 currentCorner;
```

```
private bool CurrentCornerIsFinal
{
    get
    {
        return currentCornerIndex == (path.corners.Length - 1);
    }
}
```

The NavMeshPath stores the path as an array “corners”. Each “corner” in the array is just a Vector3 for a point along the path. To make them move along the path, we just need to move them along these points, from index 0 to the last index in the array. We’ll store an int for the current index of the corner we’re in, as well as a Vector3 for the current corner so we don’t have to get it from the array every time.

We also declare a simple property that provides a shorthand to test if the current corner is the last one in the “path.corners” array.

We’ll change our Start method to add a line that initializes the current corner as the first one in the array:

```
protected override void Start()
{
    base.Start();
    currentCorner = path.corners[0];
}
```

Then we’ll implement an Update method to move and point toward the current corner. Once we’ve reached the corner, we check the CurrentCornerIsFinal property to see if we just reached the last corner in the array. If so, we Leak(). If not, we GetNextCorner() which we’ll also declare before the Update method:

```
private void GetNextCorner()
{
    //Increment the corner index:
    currentCornerIndex += 1;

    //Set currentCorner to corner with the updated index:
    currentCorner = path.corners[currentCornerIndex];
}
```

```

void Update()
{
    //If this is not the first corner,
    if (currentCornerIndex != 0)
        //Point from our position to the current corner position:
        trans.forward = (currentCorner - trans.position).normalized;

    //Move towards the current corner:
    trans.position = Vector3.MoveTowards(trans.position, currentCorner, moves
    peed * Time.deltaTime);

    //Whenever we reach a corner,
    if (trans.position == currentCorner)
    {
        //If it's the last corner (positioned at the path goal)
        if (CurrentCornerIsFinal)
            Leak();
        else
            GetNextCorner();
    }
}

```

The rest of it we've seen before already. The reason we only point at the corner if it is not the first one is because the first corner will be at the position of the spawn point, where we are already. Trying to point at a position that's exactly where we already are can cause some odd flipping behavior, so we avoid that with the first "if" in the Update call.

To implement the GroundEnemy, just use our test enemy from before, but remove the Enemy component and add a GroundEnemy instead.

If you need to make it from scratch

- Create an empty GameObject named Ground Enemy. Change it to the Enemy layer.
- Add a Box Collider sized (5, 6, 5) with a center of (0, 3, 0).
- Add a kinematic Rigidbody.
- Add a Cube sized and positioned the same as the box collider: a scale of (5, 6, 5) and a center of (0, 3, 0).

- Add a GroundEnemy script instance and set its Max Health to 12, Health Gain Per Level to 2, and Movespeed to 22. Set the “trans” reference to the root Transform, but set the “projectileSeekPoint” reference to the child Cube Transform.
- Apply the Enemy material with a hex value of DD1717.

Then, create a prefab for the Ground Enemy, and don’t forget to reference that prefab in the Player script’s groundEnemyPrefab variable.

Let’s implement flying enemies while we’re at it. They’ll be even simpler than the ground enemies. Create a FlyingEnemy script:

```
public class FlyingEnemy : Enemy
{
    [Tooltip("Units moved per second.")]
    public float movespeed;

    private Vector3 targetPosition;

    protected override void Start()
    {
        base.Start();

        //Set target position to the last corner in the path:
        targetPosition = GroundEnemy.path.corners[GroundEnemy.path.corners.
        Length - 1];

        //But make the Y position equal to the one we were given at start:
        targetPosition.y = trans.position.y;
    }

    void Update()
    {
        //Move towards the target position:
        trans.position = Vector3.MoveTowards(trans.position, targetPosition,
        movespeed * Time.deltaTime);
    }
}
```



```

        //Leak if we've reached the target position:
        if (trans.position == targetPosition)
            Leak();
    }
}

```

This makes for a similar but simplified script. It simply uses the last corner in the `GroundEnemy.path` as its “targetPosition”, set once on Start. It overrides the `targetPosition.y`, which would normally be on the ground level, to instead set it to the Y position that the Player script spawned the `FlyingEnemy` at. This keeps our Y position unchanged when our Update method moves us toward the `targetPosition`.

Once it reaches the target position, it calls `Leak()`.

Setting up the prefab is similar to the ground enemy, but we’ll add a second cube to act as somewhat crude-looking “wings”:

- Create an empty `GameObject` named `Flying Enemy` and set its layer to `Enemy`.
- Add a `Box Collider` with a size of (3, 3, 3), add a kinematic `Rigidbody`, and add a `FlyingEnemy` script. Set the “trans” reference and the “projectileSeekPoint” both to the root `Transform`. Set `Max Health` to 8, `Health Gain Per Level` to 3, and `Movespeed` to 19.
- Add a `Cube` child named `Body`. Remove its collider and set its scale to (3, 3, 3).
- Add another `Cube` as a child of `Body`. Name it `Wings`, remove its collider, and set its scale to (3, .15, 1)
- Apply the `Enemy` material to both cubes.

When you’re done, create a prefab out of the `Flying Enemy` and delete the instance from the scene. Your enemy should look something like a plus symbol, shown in [Figure 30-8](#).

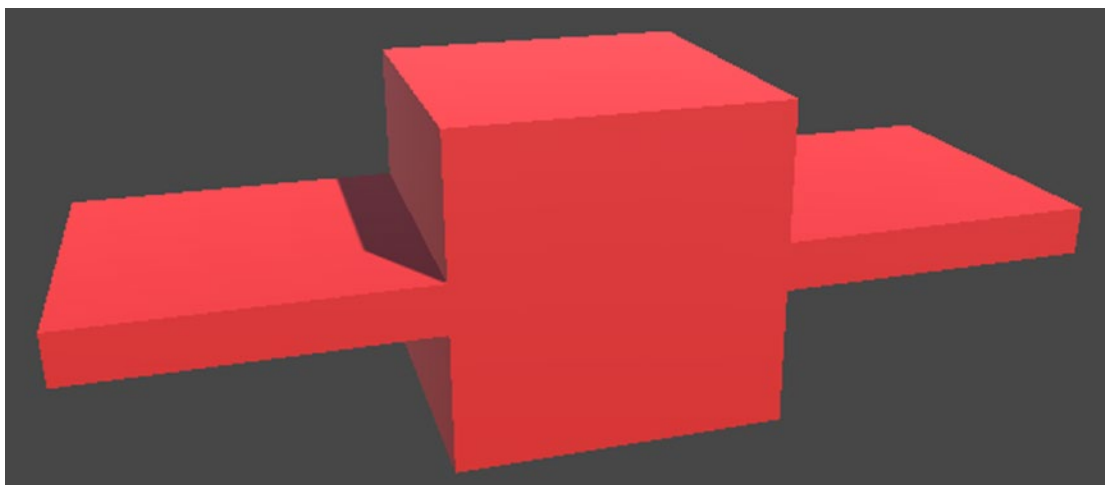


Figure 30-8. *The Flying Enemy*

With both prefabs referenced, test out our newly implemented features! Figure 30-9 shows a depiction of a ground level and flying level.

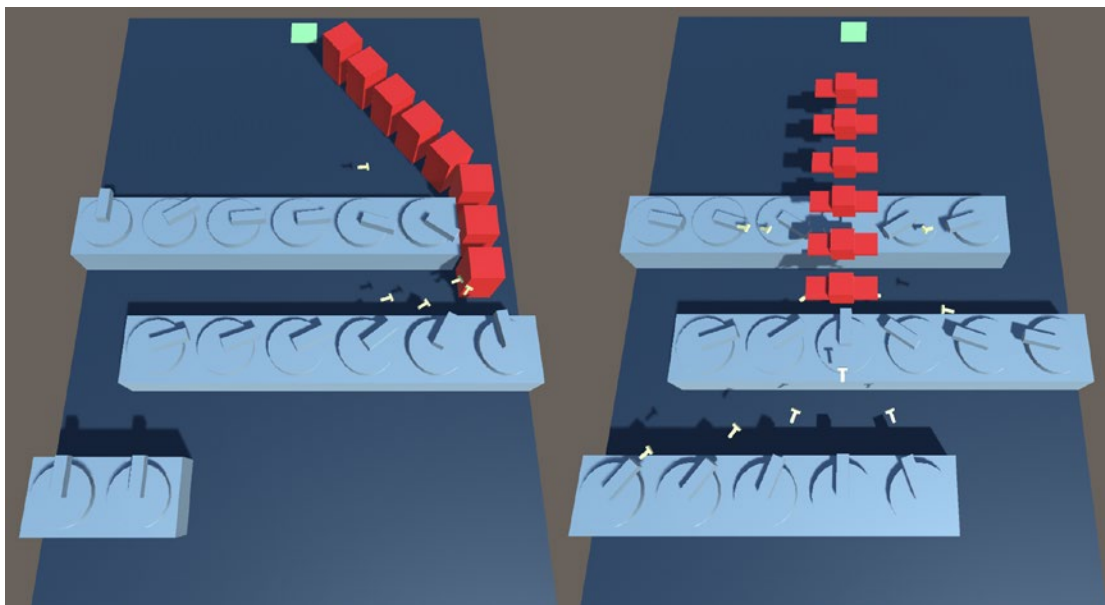


Figure 30-9. *A level with arrow towers firing at ground enemies (left) next to a level with arrow towers firing at flying enemies (right)*

Figure 30-10 shows the Game Over screen when we've lost. It waits until all enemies are gone to end the game, so that the user can see how many lives they would've needed to stay in the game.



Figure 30-10. *Game Over screen, depicting a loss with -10 lives at level 5*

Summary

This chapter got our core mechanics in place at last. We learned how to run basic pathfinding for our ground enemies whenever a tower is built or sold and how to lock the Play button from being pressed by rendering a panel over it when pathfinding fails to find a route through the maze. We also implemented both of our enemy types and completed the game loop of transitioning from build mode to play mode and then back again.

Some points to remember are as follows:

- Meshes in your game world that you want to pathfind over must be marked as **Navigation Static** through the Navigation window. You'll need to bake the settings in the Navigation window after making changes.

- **NavMeshObstacle** components with the “**Carve**” checkbox ticked will carve a hole out of the NavMesh where they are, preventing the pathfinding from going over that space.
- A **static** variable in a class will be a single instance attached to the class itself, rather than each instance of the class having their own copy of the variable.
- **NavMesh.CalculatePath** calculates a path between two points. It takes four parameters: the start point of the search, the end point of the search, an area mask resembling which areas are valid to walk over, and a **NavMeshPath** to fill with the path data.
- **NavMeshPath.status** returns a **NavMeshPathStatus** enum that can be used to determine if a path made it all the way to the end point. It will be “**PathComplete**” if it reaches the end point or “**PathPartial**” if the way was blocked and it could not reach the end.
- **InvokeRepeating** can be used within a script to continuously Invoke a script method at a given rate, only stopping when **CancelInvoke** is called.

CHAPTER 31

More Tower Types

At last, it's time to implement our final three tower types: Cannon Towers, for which we'll need to implement a projectile that arcs; Hot Plates, which will be surprisingly easy; and Barricades, which will be even easier yet.

Arcing Projectiles

Our Cannon Tower projectile is intended to curve downward as it travels toward its mark. It's far from a grand flourish, but it will give a little extra sense of weight to our so-called cannonballs.

The projectile will use its "speed" value to determine how fast it travels toward its target on the X and Z axes – outward from the tower. Its Y axis will be handled separately, following a curve from the spawn position to the floor over the duration it takes to reach the destination on the X and Z axes.

Let's create a new `ArcingProjectile` script. Make it inherit from `Projectile`:

```
public class ArcingProjectile : Projectile
```

And start off with these variables:

```
public Transform trans;
```

```
[Tooltip("Layer mask to use when detecting enemies the explosion will affect.")]
```

```
public LayerMask enemyLayerMask;
```

```
[Tooltip("Radius of the explosion.")]
```

```
public float explosionRadius = 25;
```

```
[Tooltip("Curve that should go from value 0 to 1 over 1 second. Defines the curve of the projectile.")]
```

```
public AnimationCurve curve;
```

```
//Position we're aiming to travel to. Will always have a Y value of 0.
private Vector3 targetPosition;

//Our position when we spawned.
private Vector3 initialPosition;

//Total distance we'll travel from initial position to target position,
not counting the Y axis.
private float xzDistanceToTravel;

//Time.time at which we spawned.
private float spawnTime;

private float FractionOfDistanceTraveled
{
    get
    {
        float timeSinceSpawn = Time.time - spawnTime;
        float timeToReachDestination = xzDistanceToTravel / speed;

        return timeSinceSpawn / timeToReachDestination;
    }
}
```

I'll let the comments and tooltips speak for themselves and explain just the stuff that's new.

The `AnimationCurve` is a built-in class that's going to resemble the curve the projectile takes to reach its destination. Unity has a special little popup editor that lets us set up the curve ourselves using visual tools. We can then reference our `AnimationCurve` in our code and call its `Evaluate` method, passing in a float. That float resembles the time, anywhere from the start of the curve to the end, and the method returns the value that the curve has at that point. You'll see how the curve looks in a moment, and we'll go over how it works in more detail. We'll be using it to resemble the arc the projectile takes.

The `FractionOfDistanceTraveled` member is a shorthand means of getting a value from 0 to 1 resembling how much of the distance toward our target we have traveled so far. `timeSinceSpawn` is the number of seconds that have passed since the projectile spawned. `timeToReachDestination` is the total distance we need to travel (X and Z axes only) divided by the units we travel per second, which results in the number of seconds it

takes to reach our target position. Thus, we divide the time since we spawned by the time we expect to take to reach the destination.

We'll override the protected abstract `OnSetup` method given in `Projectile`. As we established before, this method is automatically called after the `Projectile` is set up with its speed, damage, and target enemy variables. We'll use it to set up the private variables we'll use to get things done:

```
protected override void OnSetup()
{
    //Set initial position to our current position, and target position to
    //the target enemy position:
    initialPosition = trans.position;
    targetPosition = targetEnemy.trans.position;

    //Make sure the target position is always at a Y of 0:
    targetPosition.y = 0;

    //Calculate the total distance we'll need to travel on the X and Z
    //axes:
    xzDistanceToTravel = Vector3.Distance(new Vector3(trans.position.x,
    targetPosition.y, trans.position.z), targetPosition);

    //Mark the Time.time we spawned at:
    spawnTime = Time.time;
}
```

To set `xzDistanceToTravel`, we use a simple `Vector3.Distance` call, but since we don't want the Y position of the projectile to play into the equation, we just create a new `Vector3`, giving it the X and Z values from our `Transform`, but leaving its Y position at 0.

Since the `targetPosition` Y value is set to 0 just beforehand, we know that they're both equal on the Y axis, meaning it won't affect the distance between them.

Save that code and head to the Unity editor. Let's set up the projectile that will hold the script and check out this `AnimationCurve` member while we're at it:

- Create a Sphere and name it Cannon. Set its scale to (4, 4, 4) and put it in the `Projectile` layer.

- Remove the Sphere Collider component and attach an ArcingProjectile script component. Set the “trans” reference to the Transform component. For the Enemy Layer Mask, check only the Enemy layer and leave the others unchecked.
- Create a material for it if you want, or just use the Arrow material to make it the same color as the Arrow Tower projectiles.

When you click the “curve” field, a popup Curve editor will show. Inside this editor a graph of sorts is visible, which is where the curve will be laid out – although there won’t be a curve at all by default. The buttons on the bottom of the editor are presets which can be clicked to fill the graph in with a curve. Click the third preset from the left so that your curve editor looks like Figure 31-1.

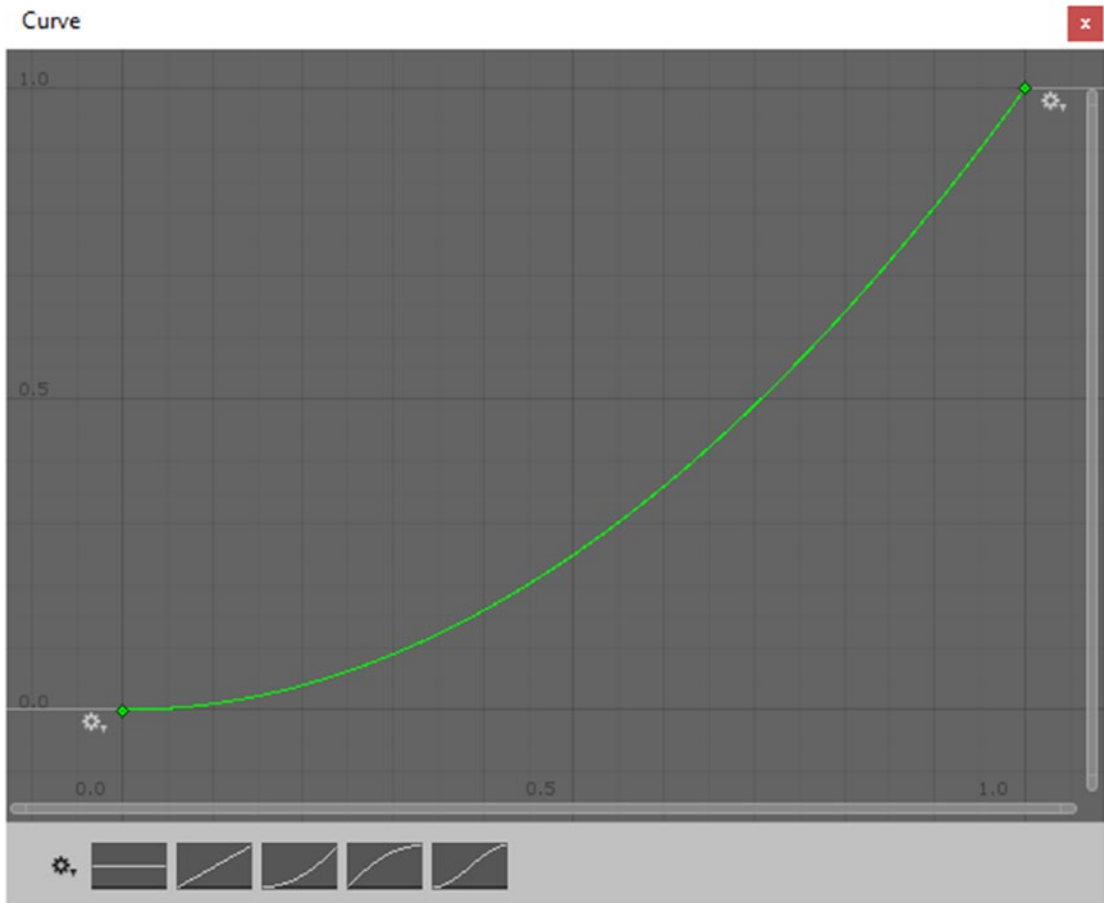


Figure 31-1. The Curve editor with the third preset selected

The horizontal axis (left to right) resembles time, and the vertical axis (up and down) resembles the value. As you can see, each axis has number values on the bottom, showing what the time or value is at that point on the axis.

The bottom left is 0 value and 0 time. The top right is 1 value and 1 time.

To make use of the curve, we simply pass in a time and get back the value corresponding to that time. The equivalent of this, in person, is to place your finger somewhere on the bottom axis, like on the 0.5 mark (halfway through the curve). Then, trace your finger up until it touches the green line of the curve. After that, just trace your finger straight left until you hit the vertical axis. Whatever the value is at that point, that's your result.

That's what the `AnimationCurve.Evaluate` method does: we give it a float parameter for the time, and it gives us the corresponding value.

That preset is exactly the value we want for our curve. We'll use the `Evaluate` return value as the fraction to `Lerp` the Y value of the projectile toward the Y value of the target position (which will be 0, the floor level). It will start at 0 (the bottom left of the curve), and over time, it will raise to a value of 1.

Don't forget to create a prefab out of the Cannon and remove it from the scene. Now let's declare our `Update` method and see how this `Lerp` call is going to work:

```
void Update()
{
    //First, we'll move along the X and Z axes.
    //Get the current position and zero out the Y axis:
    Vector3 currentPosition = trans.position;
    currentPosition.y = 0;

    //Move the current position towards the target position by 'speed' per
    second:
    currentPosition = Vector3.MoveTowards(currentPosition, targetPosition,
    speed * Time.deltaTime);

    //Now set the Y axis of currentPosition:
    currentPosition.y = Mathf.Lerp(initialPosition.y, targetPosition.y,
    curve.Evaluate(FractionOfDistanceTraveled));

    //Apply the position to our Transform:
    trans.position = currentPosition;
```

```

    //Explode if we've reached the target position:
    if (currentPosition == targetPosition)
        Explode();
}

```

First, we get the position of our Transform in a new Vector3. We'll apply this value to the Transform after we've modified it. We set its Y position to 0 and then use MoveTowards to travel only on the X and Z axes toward the target position.

Then we handle the Y axis. Each frame, we set our Y position to the result of a Lerp call starting at the initial Y position (when we first spawned) and moving toward the target Y position (the floor). Thus, we want the fraction (third parameter) to start at 0, our position, and raise to 1 (the floor position) over the duration it takes to reach the target position. If we just passed in FractionOfDistanceTraveled, it wouldn't curve, though. It would take a straight line. So we pass that value into the curve.Evaluate method. With the time going from 0 to 1 over the duration of the projectile's path, it works out perfectly: the curve takes our Lerp value from 0 to 1 in a fancy way.

But before we can test, we need to declare the Explode method to damage enemies in a radius:

```

private void Explode()
{
    Collider[] enemyColliders = Physics.OverlapSphere(trans.position,
        explosionRadius, enemyLayerMask.value);

    //Loop through enemy colliders:
    for (int i = 0; i < enemyColliders.Length; i++)
    {
        //Get Enemy script component:
        var enemy = enemyColliders[i].GetComponent<Enemy>();

        //If we found an Enemy component:
        if (enemy != null)
        {
            float distToEnemy = Vector3.Distance(trans.position, enemy.
                trans.position);
            float damageToDeal = damage * (1 - Mathf.Clamp(distToEnemy /
                explosionRadius, 0f, 1f));

```

```

        enemy.TakeDamage(damageToDeal);
    }
}

Destroy(gameObject);
}

```

Here, we use a new method: `Physics.OverlapSphere`. This tests for collisions against colliders in a sphere at the position given in the first parameter, sized by the radius given in the second parameter, and using a layer mask given in the third parameter. That's where we use our enemy layer mask to make sure we're only getting `Enemy` instances. It returns an array containing all the `Colliders` touched by the sphere.

We then loop through these colliders, grab their `Enemy` component, and calculate the distance from the projectile to the `Enemy`. Using that, we can calculate a value from 0 to 1 resembling how far the enemy is from the center of the explosion radius, where 0 is right in the middle of the projectile and 1 is at the very edge. We want to do full damage to enemies in the center and less as they grow further from the center, so we need to “flip” the value by subtracting it from 1. That gives us a multiplier for the damage to deal.

Of course, we deal the damage to the enemy with the `TakeDamage` method. After doing this for each touched enemy, we `Destroy` the projectile.

Cannon Tower

Let's get our Cannon Tower prefab ready to test our `ArcingProjectile` script:

- Create an empty `GameObject`. Name it `Cannon Tower` and place it in the `Tower` layer. Give it a `NavMeshObstacle` component with a size of (10, 6, 10) and a center of (0, 3, 0). Make sure to check the “Carve” box.
- Add a `Cube` child named `Base`. Scale it to (10, 6, 10) and position it at (0, 3, 0).
- Add a `Sphere` child to the root `Cannon Tower GameObject`. Position it at (0, 6, 0) and scale it to (7, 7, 7).

- Add a Cylinder child to the root Cannon Tower GameObject. Name it Barrel and give it a 90 degree X rotation to point it forward. Give it a position of (0, 7.5, 3) and a scale of (2, 1, 2). Make it a child of the Sphere so it turns when the Sphere turns.
- Add an empty GameObject as a child of the Barrel. Name it Projectile Spawn Point and give it a position of (0, 1, 0) to put it at the end of the barrel. To point its forward axis out from the barrel, give it an X rotation of 270.
- Open your Arrow Tower prefab and locate the Targeter we made for it. Copy it with Ctrl+C or by right-clicking and clicking Copy in the context menu. Return to the scene and paste the Targeter, make it a child of the root Cannon Tower GameObject, and set its local position to (0, 0, 0).
- Add a FiringTower script component to the root Cannon Tower GameObject. Set the references. The “aimer” should be the Sphere, and the projectile prefab should be the Cannon we created earlier.
- Set the Gold Cost to 8, Range to 30, Fire Interval to .75, Damage to 9, and Projectile Speed to 80.
- Apply the Tower material to all the individual pieces.
- Create a prefab out of the root GameObject and then remove it from the scene.

When you’re done, your Cannon Tower should look something like Figure [31-2](#).

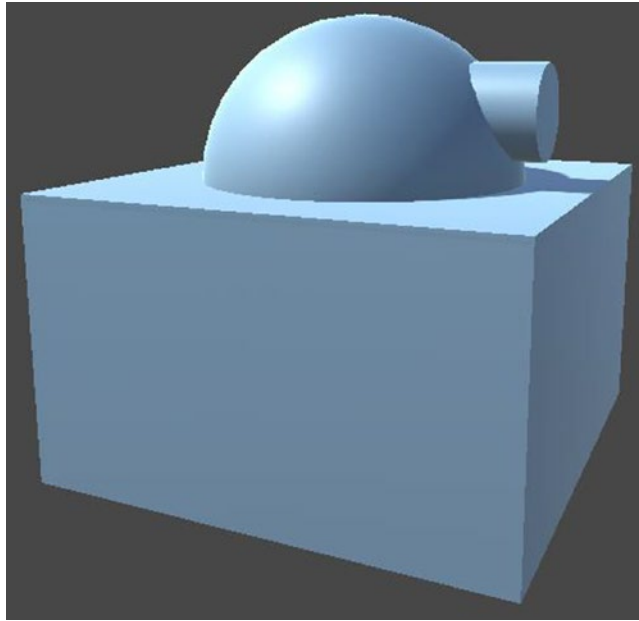


Figure 31-2. *Our Cannon Tower*

One final step: The Build Button for our Cannon Towers needs its OnClick events set up so that we can build the tower to test it.

You know how to do this. Find the Cannon Tower Build Button GameObject tucked away in the Canvas and set it up with two events, each pointing at the Player Camera GameObject. One calls `Player.OnBuildButtonClicked` and provides the Cannon Tower prefab as a parameter. The other calls `Player.SetSelectedBuildButton` and provides the Image component, just above the Button component, as its parameter. It should look like Figure 31-3.

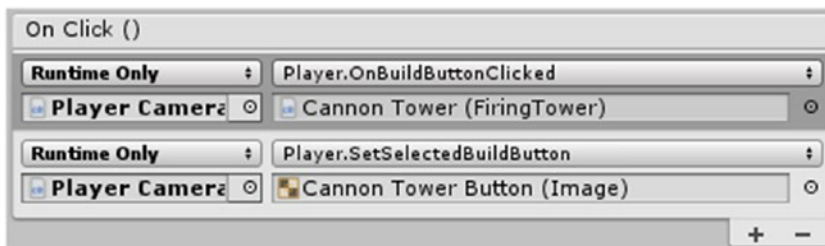


Figure 31-3. *The OnClick event in the Inspector, at the bottom of the Button component of our Cannon Tower Build Button GameObject*

With that, you should now be able to hop into the game, select the Cannon Tower build button, and place some Cannon Towers. Once you play the level, you'll see the cannon towers firing. Particularly when you watch from an angle in the Scene view, you can tell how the projectiles curve toward the ground instead of taking a straight line toward it. If you want to watch it very closely, you can click the Pause button beside the Play button in the Unity editor, get your camera in position in the Scene view, and use the hotkey Ctrl+Alt+P to "step" one frame forward at a time. Hold the keys to play out the frames in quick succession.

There's just one final step. Our Cannon Towers can still attack flying enemies. We didn't want that. Only Arrow Towers should fire at flying enemies.

To implement this, we'll go up the inheritance chain to our FiringTower script. First, add a variable up at the top of the script class that specifies if the tower can or cannot attack flying enemies:

```
[Tooltip("Can the tower attack flying enemies?")]
public bool canAttackFlying = true;
```

Before you forget, go uncheck that field in the Cannon Tower prefab so they can't attack flying enemies.

In the Update method, locate this block:

```
else //If the enemy is alive and in range,
{
    //Aim at the enemy:
    AimAtTarget();

    //Check if it's time to fire again:
    if (Time.time > lastFireTime + fireInterval)
    {
        Fire();
    }
}
```

And update it to this:

```
else //If the enemy is alive and in range,
{
    if (canAttackFlying || targetedEnemy is GroundEnemy)
    {
        //Aim at the enemy:
        AimAtTarget();

        //Check if it's time to fire again:
        if (Time.time > lastFireTime + fireInterval)
        {
            Fire();
        }
    }
}
```

We’ve added an “if” that checks whether the tower can attack flying enemies, or, if it can’t, we check that the targeted enemy is a `GroundEnemy`.

With that, our cannon towers are complete!

Hot Plates

Two tower types to go. I promised this one would be easy, and it’s a short script, so I’m going to show the whole thing at once (except for the usings). Create a script named `HotPlate` and write this code:

```
public class HotPlate : TargetingTower
{
    public float damagePerSecond = 10;

    void Update()
    {
        //If we have any targets:
        if (targeter.TargetsAreAvailable)
        {
            //Loop through them:
```

```

        for (int i = 0; i < targeter.enemies.Count; i++)
        {
            Enemy enemy = targeter.enemies[i];

            //Only burn ground enemies:
            if (enemy is GroundEnemy)
                enemy.TakeDamage(damagePerSecond * Time.deltaTime);
        }
    }
}

```

Simple enough, right? Since it inherits from `TargetingTower`, it will already have its `targeter` set up automatically, tracking enemies within. We just need to “burn” them for constant damage per second. Since the `targeter` will find flying enemies as well, we need to make sure we only burn an enemy if it’s a `GroundEnemy`.

Let’s set up the prefab and build button:

- Create an empty `GameObject` named `Hot Plate`. Put it in the `Tower` layer and give it a `Hot Plate` script. Set the `Gold Cost` to 12 and `Range` to 5. Its `damage per second` should also be at 10. The reason we set `Range` to 5 is because, if you’ll recall, we’ve set the `Range` field up to mean “how far away from the center of the Tower.” Thus, it needs to be half of the size of the plate, not the full size!
- Add an empty `GameObject` child to the `Hot Plate`. Name it `Targeter` and put it in the `Targeter` layer. Give it a trigger `Box Collider`, a `Targeter` script component with the “`col`” field set to a reference to that `Box Collider`, and a kinematic `Rigidbody`.
- Add a cube as a child to the `Hot Plate`, positioned at (0, .05, 0) with a scale of (10, .1, 10). I’ll apply a new material to mine with a bright-orange color using a hex value of `FF6034`.
- Don’t forget to set the `Targeter` reference in the `Hot Plate` script.
- Create a prefab for the `Hot Plate` and delete it from the scene.

Then, set up the build button the same way we set up the `Cannon Tower` build button – but, of course, reference the `Hot Plate` prefab.

That's it. Our Hot Plates won't have a `NavMeshObstacle` member because they are meant to be walked over by enemies, not navigated around.

Barricades

Barricades are just Towers with nothing else to them. They'll be a simple cube with no cool turret on top:

- Create an empty `GameObject` named `Barricade` and put it in the Tower layer.
- Add a base Tower script with a Gold Cost of 2.
- Once again, give it a `NavMeshObstacle` component with a size of (10, 6, 10) and a center of (0, 3, 0). Make sure to check the "Carve" box.
- Add a Cube child and give it the same position (0, 3, 0) and scale (10, 6, 10) as the `NavMeshObstacle`. You can remove the Box Collider component.
- Since they don't blow things up, I'm giving my Barricade a drab-brown color with a hex value of `A6843F`.

You know the drill. Make a prefab, delete the instance from the scene, and set up the build button.

Summary

With that, we've implemented all four of our tower types and hooked them all up to their corresponding build buttons. We learned how to use the `AnimationCurve` and how to test for all Colliders touching an imaginary sphere using the `Physics.OverlapSphere` method.

Figure 31-4 shows a little maze set up with all four tower types involved.

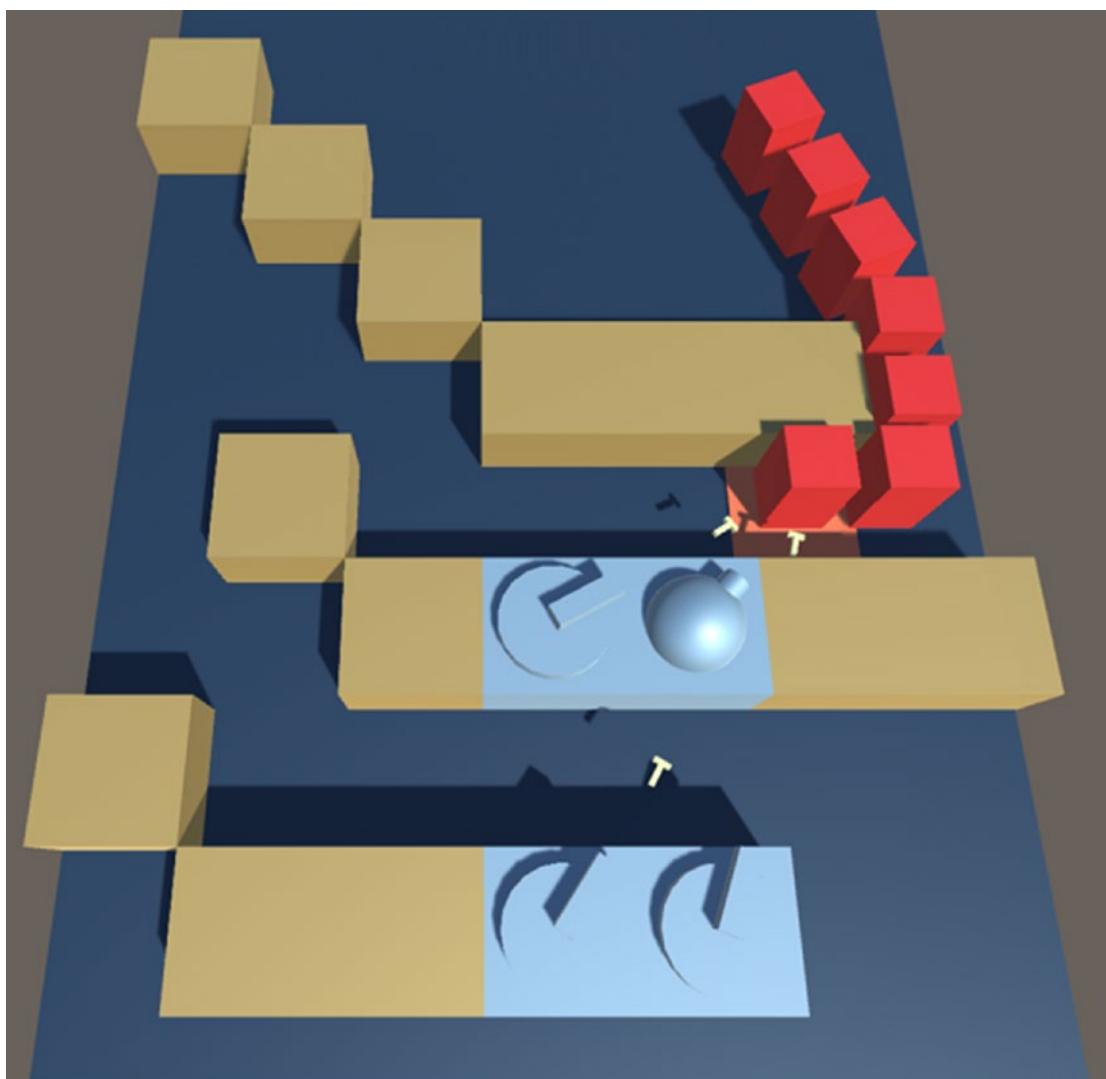


Figure 31-4. *A little maze with Barricades, Arrow Towers, a Hot Plate with some unfortunate enemies standing on it, and a Cannon Tower*

CHAPTER 32

Tower Defense Conclusion

We have at last concluded our second project. It's no finished game, but we've learned quite a lot along the way. Let's get a quick refresher on it all and then review some ideas for additional features you could try to implement on your own.

Inheritance

A major focus of this project was inheritance. We talked about it in the first part, but this project puts it to work. We used base classes like `Tower` and `Projectile` to define a set of variables and logic that can be implemented in different ways:

- A `Tower` can be bought and sold. To implement that, we need not concern ourselves with what specific type of `Tower` it is. We can reference it as a `Tower` to grab its gold cost and refund factor. We don't know what it does, but we don't need to. We just need to let the player put it on the stage or take it off and adjust the player's gold accordingly.
- A `Projectile` stores a travel speed and damage and is always spawned with a certain enemy as its target. When creating a projectile, we need not worry about whether it's an `Arcing` or `Seeking` projectile. We just give it a speed, damage, and target enemy and let the subclass do the rest.

This demonstrates one of the important takeaways of inheritance: that you can reference an upper type like `Tower` or `Projectile` even if a lower type is actually stored in the reference. If you only need the features of a `Tower`, like gold cost and refund factor,

you can look at any more specific version as a Tower. If you ever need to make the distinction of what exact type it is, you can do so with casting. We demonstrated this with the Targeter, casting the Collider type to BoxCollider or SphereCollider.

We also put abstract classes to use, which is a means of making a class that cannot have instances of itself created. These classes are never intended to be used as is, instead acting as base classes that we inherit from to get use out of them. You'll only ever create instances of the lower types of the abstract class, not the abstract class itself.

This led us to using abstract and virtual methods. You declare the method, marking it either as "abstract" or "virtual," on the upper type:

```
public class UpperType
{
    public virtual void Setup()
    {
        //[setup code for the upper type goes here]
    }
}
```

You can then declare the method on a lower type to allow it to extend the method with its own logic. This is done with the "override" keyword:

```
public class LowerType : UpperType
{
    public override void Setup()
    {
        //Call the upper type's version of the method:
        base.Setup();

        //[setup code for the lower type goes here]
    }
}
```

You'll recall that abstract methods can only be declared on abstract classes, and the abstract method declaration won't give its own code block. The lower type must declare an override for the abstract method and provide a code block for the implementation (even if it's just an empty code block). However, with virtual methods, lower types are not forced to declare an override.

UI

We put Unity's UI system to use to make something a bit more than just the bare minimum. We learned that this UI system operates through `GameObjects` and components, making use of the hierarchical structure of parents and children. All our UI elements are children of the `Canvas` and can further be nested inside each other. Using pivots, we can determine how individual elements and their children rotate and scale.

Of course, most UI requires some form of responsiveness to the user. We learned how to add functionality by responding to events, using `OnClick` events found in the `Inspector` to call methods on our `Player` script when our build buttons were pressed. You can use this to call methods or set built-in variables on a `GameObject` or any of its scripts or components. But if you're trying to call a method you declared on one of your scripts, that method must be public, and it must either have no parameters or have one parameter of a serializable type like a script, a built-in type, or a component.

Raycasting

We learned how to use `Physics.Raycast` to fire a ray that will strike any colliders of a given `LayerMask`, returning `true` if an object was hit. Using this, we detected objects under the mouse by converting mouse position to a ray shooting out of the `Camera` with the `ScreenPointToRay` method.

If you visit the official Unity scripting API for the `raycast` method, you'll see that it has many overrides allowing you to call it with different parameters. You can avoid using the `Ray` and `RaycastHit` types that we used and opt to instead simply give a `Vector3` for the ray start point and a `Vector3` for the direction it travels in:

```
if (Physics.Raycast(origin,direction))
{
    //...
}
```

If this method is used, the “`maxDistance`” parameter defaults to `Mathf.Infinity`, so the ray will travel infinitely in the direction it was cast. If you'd rather give a maximum distance yourself, you can provide a third parameter (a float).

Alternatively, you can use another similar override with an “out RaycastHit” as a third parameter and a “maxDistance” as a fourth parameter. Pretty much any combination you could need is covered!

Pathfinding

We learned how to perform some basic pathfinding to give us a series of points for our ground enemies to travel along to take them through the player’s maze. Through the Navigation window, we marked our Stage plane as Navigation Static, updated the Agent settings to match the size of our enemies, and Baked the settings to the scene. This allows us to call `NavMesh.CalculatePath` to fill an existing `NavMeshPath` instance with the points that make up the path.

By making our path a static variable in the `GroundEnemy` class, we allowed ourselves to access it from other classes without requiring a reference to an instance of a `GroundEnemy`.

Additional Features

You might already be cracking your knuckles in preparation to tweak numbers for the game, like enemy health and speed, tower damage and rate of fire, and gold costs and rewards. Those things can be fun to play with, and our current settings certainly don’t make the game particularly challenging. I’ll leave that design stuff to you and suggest some features that require a bit of coding and implementation.

Health Bars

One feature of polish we’re missing is an indication of enemy health. It’s not very satisfying to watch our cannon towers shooing at our enemies when we can’t even see their damage “splashing” to nearby enemies! The obvious answer would be to give each enemy a health bar above its head.

You can do this with world space UI. You’ll need to add a separate Canvas to the scene and change its Render Mode to World Space. You’ll also want to change the “Reference Pixels Per Unit” field of the Canvas Scaler component to something lower, like 10.

Health bars can then be positioned in world space, but they still must be children of the world space Canvas, or they won't render. Since health bars can't be children of their corresponding Enemy, you'll have to make the Player script create a health bar prefab instance for each Enemy when it is spawned and then script the health bars to position themselves above the enemy's head every frame and to automatically destroy themselves when the enemy dies. Alternatively, you could put a health bar prefab instance in the enemy prefabs, so they spawn with their own health bar and then script the bar to make itself a child of the World Canvas after – but this means you'd have to place a health bar on each enemy prefab separately.

You can use a Panel as a dark-red background for the health bar and then use a bright-red Panel inside that one as the “fill.” Size the “fill” to completely cover the background. Every frame, set the X axis of the “fill” scale to the percentage of life remaining on the enemy the health bar is associated with: `enemy.health/enemy.maxHealth`. That would be done with a script attached to the health bar, with a reference to the `RectTransform` of the “fill” panel. Once the enemy loses health, the fill begins to shrink, but the background panel will be there to show behind it.

You can determine how the fill panel shrinks by setting its origin with the rect tool. The origin is that blue circle we talked about, which can be clicked and dragged. If you leave the origin in the center of the panel, the shrinking will pull the left and right sides of the panel in toward the center until nothing remains. If you put it at the left side, then the right side will shift toward the left until nothing remains.

Types for Armor and Damage

Some tower defense games make use of differing types of armor and damage. Each level, enemies will use a different armor type from a selection of, say, three, like metal, wood, or magic. Each tower could then be assigned a damage type, and each damage type is strong against certain armor types, but weak against others. This encourages the player to have a range of towers that deal damage in all the types so that no enemies are particularly difficult to handle.

More Complex Pathing

Make the ground enemies touch little points on the stage along the way to the leak point. Rather than running from the spawn point to the leak point, put a few extra points

between the two, designated by similar colored plates (cylinders). For example, enemies could go from the spawn point to the first plate, then to the second plate, and then to the leak point.

This is a fun little mechanic that gives the player something to maze around. If you know your enemies will have to touch certain spots on the stage before they continue, you can maze laboriously around those points and place your most important towers within range of them to ensure they get used as much as possible. Enemies will have to navigate through the maze, touch the point, and then navigate back out. It adds an extra layer of tactics to the mazing concept.

To implement this, ground enemy pathfinding would have to be changed. You can't just pathfind from the spawn point to the leak point anymore. You could implement this with a `List<Vector3>` to store all the points. Pathfind from the spawn point to the first plate. Add the corners to the List. Then, pathfind from the first plate to the second plate, and add those corners to the List. Then, pathfind from the second plate to the leak point, and add the corners.

That can be done using the List instance method "AddRange". It adds the items from an array or List to the end of another List, for example:

```
var points = new List<Vector3>();

// [Perform pathfinding]

//Add points:
points.AddRange(path.corners);
```

This adds the path corners to the "points" List.

You would also need to make sure that towers can't be built directly on top of the points. Since enemies must touch the point, placing a tower on top of it makes the path impossible. Each point should be centered at a position a multiple of 10, placing it directly in the slot a tower would normally go. Then you can update the tower building logic to not allow placing towers when the highlighter is on top of one of these points.

Range Indicators

Give the player some indication of the range that firing towers have. You could make a thin Cylinder with a semi-transparent material, named Range Highlighter. When the player puts their mouse over a Tower (detected with a raycast), make that

the highlighted tower. Whenever the highlighted tower changes, make sure it's a `FiringTower` by performing a cast. If so, size the range highlighter `Cylinder` to match the tower's range and center the `Cylinder` on the tower. If it's not a `FiringTower`, hide the range highlighter by deactivating it.

Upgrading Towers

Give the player a means to upgrade existing towers, paying some gold to make them stronger. You could change the Tower Selling Panel to give it an “Upgrade” button and some text for the tower name and current level. When the player upgrades a tower, charge them some gold and strengthen some of the tower's stats based on what type of tower it is: damage, projectile speed, and range could all rise. Or you could make the player upgrade individual stats and track the level of each stat separately.

Summary

Now that we've gone over the important stuff you've learned throughout this project and given you some ideas for additional features to implement, it's up to you to decide if you want to linger on this project and try to expand it yourself or move on to our next project to continue the book. This project has taken us a long way with programming fundamentals like inheritance and working with collections like `Lists` and `Dictionaries`. In the next project, we'll be dealing with physics and 3D movement systems more in-depth.

PART III

Physics Playground

CHAPTER 33

Physics Playground Design and Outline

For our third example project, we'll be tackling some new topics pertaining to the physics of the Unity engine. We'll implement a mouse-aimed camera with 3D movement for our player, including jumping, gravity, and wall jumping. We'll play with some objects using proper Rigidbody-controlled physics, giving our player a means of pushing and pulling them from a distance. We'll also tinker with joints to attach Rigidbodies to each other and "force fields" to play with adding forces to Rigidbodies and/or our player on the fly.

Feature Outline

This project will play less like a game and more like a testing ground. We'll get our hands dirty with different aspects of the physics engine, as well as working with fully 3D player controls, since up until now our projects have used top-down camera angles.

Camera

Our camera will have two modes: first-person and third-person. You're probably comfortable with these terms already. First-person is "through the eyes of the character," turning left, right, up, or down as the mouse moves. Third-person is "over the shoulder," hovering behind the character and orbiting around them as the mouse moves.

We'll allow the player to switch between these modes on the fly with the press of a hotkey, smoothly moving the camera from one mode into the other. We'll have proper smoothing applied to our camera's rotation, configurable in the Inspector to allow us to select how much smoothing we want. This not only makes first-person rotation smoothly respond to the mouse, but also the third-person orbiting.

The player will also be able to change how far the third-person camera will hover behind their character, using the scroll wheel of their mouse to draw the camera closer

or pull it further away. Of course, we'll limit this to a certain minimum and maximum distance to keep the player from going crazy with it.

Player Movement

Since we have a mouse-aimed camera, we'll be implementing a "more 3D" movement system than our first project: the WASD keys to move local to the direction the camera is pointing, Space while grounded to jump, and Space while midair to attempt to "wall jump" off of a nearby surface. Once we go midair, whether by jumping or running off a ledge, we'll carry with us any ongoing velocity. Once we become grounded again, that velocity will drag out over time, if we aren't using WASD to move.

Wall jumps are performed by pressing Space while midair with a surface anywhere near the sides of our character. It can be behind, in front, left, or right; it doesn't matter. It just has to be close enough.

A wall jump will provide upward and outward momentum. If we're not holding any WASD keys when we perform a wall jump, it just goes straight up. If we are holding WASD keys, we'll also "push off" in that local direction – for example, holding W will move us forward as well as upward.

To implement this, we'll be using a different method of tracking our player velocity, employing only one velocity variable that handles velocity given by movement as well as external forces like force fields pushing us. This will demonstrate a handful of new and useful concepts for working with vectors.

Pushing and Pulling

To experiment with applying forces to Rigidbodies, we'll give our player a "telekinesis" power, allowing them to point at an object that has a Rigidbody and either hold left-click to pull the object toward them or hold right-click to push it away from them.

This power will be limited so that it only works on objects that are close enough to the player. We'll draw a simple four-pixel square at the center of the screen to demonstrate where the mouse is pointing and change its color to respond to what the telekinesis is currently doing. It will be gray when the player is not pointing at something that can be affected by telekinesis, white if the player is pointing at something valid that's in range, orange if the object is valid but not in range, and green while we are actively pulling or pushing something.

Moving Platforms

By default, our player will simply remain still even if the object they're standing on is moving. If you want to have floating platforms that move around, you probably want the player to move with the platform. We'll code up a means of setting up a platform that other objects will "attach" to when they land on it. With that in place, we'll code a script that makes a platform move back and forth between two points to exhibit that the player moves with it.

Swings

We'll learn how to create a series of linked objects, assembling something like a chain to hold up a swinging platform. The player can use their telekinesis to push and pull the platform to make it swing and can stand on it while it's moving. They can even apply telekinesis to individual links in the swing.

Force Fields and Jump Pads

We'll implement two similar systems: force fields that constantly add velocity in a given direction to all objects that remain within them and jump pads that apply a sudden change in velocity to any object when it first touches the pad. Both of these systems can be adjusted to make them work on miscellaneous GameObjects, the player, or both. Since the player velocity is handled by our own script, not a Rigidbody, we'll have to react differently when the player touches the field as opposed to an object controlled by a Rigidbody.

Project Setup

Let's get our project ready before we begin. With Unity Hub, create a new project using the 3D template. I'll name mine "PhysicsPlayground".

In Edit ► Project Settings ► Tags and Layers, we'll set up three layers:

- **8: Player** – Only the player character will use this layer.
- **9: ForceField** – Jump pads and force fields will use this layer.

- **10: Unmovable** – We’ll make the player’s “telekinesis” work on all layers except this one. If we ever want a GameObject that’s controlled by a Rigidbody, but can’t be pulled and pushed by the player, we can put it in this layer.

When you’re done, your layer settings should look like Figure 33-1.

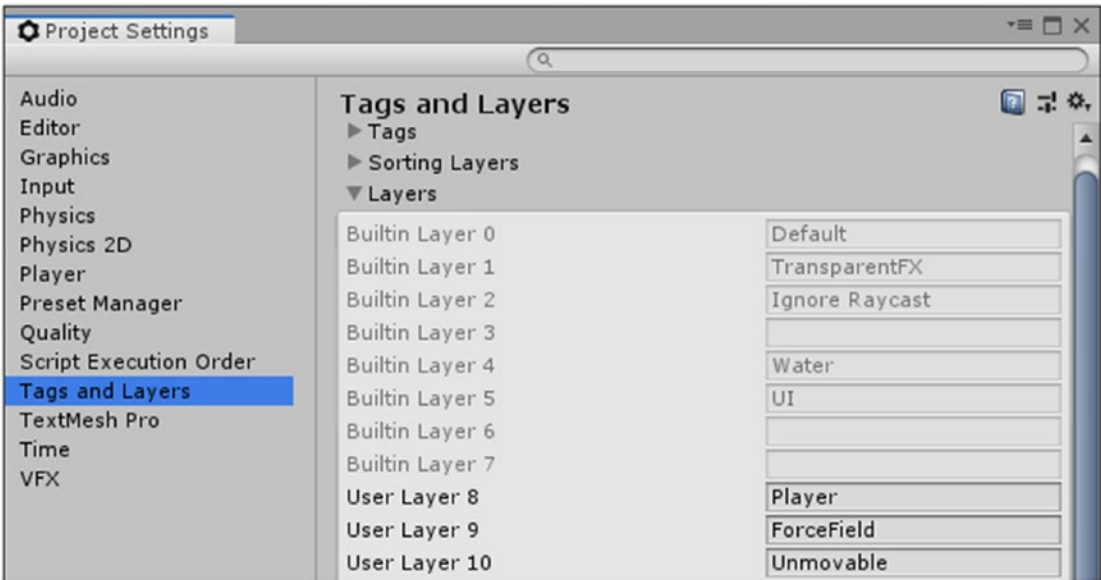


Figure 33-1. Our layer settings in the Edit ► Project Settings window

We’ll also rename our default scene from “SampleScene” to “Main” and create these folders in the Assets folder through the Project window:

- Materials
- Prefabs
- Scripts

Summary

With a general idea of how our project will play and the features we expect to implement, let’s press on and start adding mechanics one by one. When we’re done, we’ll have hands-on experience with most of the major concepts and components of Unity’s built-in physics.

CHAPTER 34

Mouse-Aimed Camera

Before we give the player any means of moving around, we'll have to set up the player `GameObject` and provide them with a way of changing the direction they're facing. We'll do that here, implementing our first- and third-person cameras. Both systems will be handled by one script, and the player will be able to press a hotkey to smoothly switch from one to the other.

Coding the first-person camera will be a bit simpler. We position it where the player's eyes would be, and we rotate it based on changes in the mouse position. Move the mouse left, the camera and the player will turn left, and so on.

The third-person camera requires some extra legwork. We want it to be positioned a certain distance behind the player at all times, orbiting around the player as the mouse moves, but always looking at a certain point on the player. We also want to prevent it from passing through walls in its way, so that it instead slides along the surface of the wall.

Player Setup

Let's get the player all set up in the scene:

- Create an empty `GameObject` named `Player`. Position it at (0, 0, 0). Apply the `Player` layer to it.
- Add a child empty `GameObject` to the root `Player` `GameObject`. Name it `Model Holder`. Remember, you can right-click the `Player` and select "Create Empty" to add a child directly to it. It should automatically have the `Player` layer. Keep its local position at (0, 0, 0).
- Add a `Capsule` child to the `Model Holder`. Set its local position to (0, 3, 0) and scale it to (2, 3, 2). This makes it 6 units tall and puts its bottom right at the root `Player` position.

- Remove the Capsule Collider component from the Capsule. We'll be using a CharacterController for collisions instead, which will be set up in the next chapter.
- Add another empty GameObject as a child of the root Player GameObject. Name it Camera X Target. Leave its local position at (0, 0, 0). We'll just be using it for its Transform.
- Copy and paste the Camera X Target and rename this new instance to Camera Y Target.
- In the Hierarchy, locate the "Main Camera" GameObject that's included in the scene by default. Drag it onto the Player to make it a child of the root Player GameObject, just like the Model Holder and the Camera Targets. If you've deleted the Main Camera, you can just right-click the Player and create a new Camera instead. Change its layer to Player. Name it Player Camera. In the "Clipping Panes" field, set the "Near" value to 0.01 instead of the default value of 0.3. Set its local position to (0, 5.4, 0), placing it up near the top of the player.

How It Works

The root Player GameObject holds all of the others. This one never rotates. It should remain at (0, 0, 0) rotation at all times. The Model Holder is what we apply rotation to, keeping it facing forward along the facing direction of the camera, but only on the Y rotation axis, which is what turns it left and right. We don't want to affect its X rotation, which would tilt it up toward the sky or down toward the ground. If you were working with a player model, you wouldn't want their whole body to tilt up and down when they raise or lower their head, right? You would, however, likely want to have their upper body bend forward or back as they look down or up, but our player is just a capsule, so we won't be getting into that here.

The two Camera Target GameObjects will play several important roles in our camera system. You can effectively look at them as one entity, the Camera Target, but we have them as two separate Transforms, one for X rotation and one for Y rotation, to avoid problems down the road that would occur if we applied both axes of rotation to the same Transform. When we use their rotation, we'll add the X and Y rotation together.

Let's detail the purpose of our Camera Target:

- Whenever the player moves the mouse, we'll immediately apply that rotation to the Camera Target, keeping it pointing in the direction the first-person camera should be pointing. X rotation is applied to Camera X Target, and Y rotation is applied to Camera Y Target. If we're in first-person mode, the actual camera will have its rotation Slerped toward that of the Camera Target to generate an effect of smoothing to the first-person camera movement.
- The target position of the third-person camera is determined by raycasting directly backward from the Camera Target rotation. If that ray hits a wall, the target position is at the "hit.point" where the ray struck the wall. Otherwise, it's at the end of the range of the ray, which is determined by a variable we can change in the Inspector. The third-person camera will always move toward the target position, using Lerp so we can add smoothing if we want.

As you can see, our whole process revolves somewhat around the Camera Target. Every frame, we'll check mouse movement and rotate the Camera Target first based on that: left and right mouse movement applies to the Camera Y Target, while up and down mouse movement applies to the Camera X Target. Then we run logic based on whether we are in first- or third-person mode. Either way, that logic is going to rely on the Camera Target to determine how the camera moves and rotates: first-person mode just smoothly follows the rotation of the Target, and third-person mode determines its target position based on the Target facing and then Lerps toward that position.

The Camera Target position is not important, though. When we cast the ray to determine our third-person camera target position, we don't actually originate the ray at the Camera Target position, so it doesn't matter where we place the Target. All we use it for is its facing direction.

We'll instead provide ourselves with a Vector3 variable that we can use to set an orbit point for the third-person camera. This is the point around which the third-person camera spins as the mouse moves. That means it's the origin point for the ray we'll be casting to determine the target position.

This orbit point will be local to the Model Holder so that as the Model Holder rotates with the third-person camera movement, the orbit point remains at the same position relative to it. This allows us to place the orbit point somewhere off to the side of the character model, such as beside one of their shoulders, and the point will always remain relative to the model rotation.

This orbit point is also the point that the camera will be pointed toward (the point it will “look at”).

Script Setup

Let’s get to it. In your Project, right-click your Scripts folder and select Create ► C# Script. Name the new script “PlayerCamera”.

Let’s start off with the variables, and we’ll be using Start, Update, and LateUpdate events:

```
public class PlayerCamera : MonoBehaviour
{
    //References:
    [Header("References")]
    [Tooltip("The base Transform of the player, which should never rotate.")]
    public Transform playerBaseTrans;

    [Tooltip("Set this to the Transform that has the Camera component
    (which should also have the PlayerCamera component).")]
    public Transform trans;

    [Tooltip("Reference to the Camera X Target Transform.")]
    public Transform cameraXTarget;

    [Tooltip("Reference to the Camera Y Target Transform.")]
    public Transform cameraYTarget;

    [Tooltip("The Transform holding the model. This is what will rotate on
    the Y axis to turn left/right as the camera turns.")]
    public Transform modelHolder;

    //Movement and Positioning:
    [Header("Movement and Positioning")]
    [Tooltip("How quickly the camera turns. This is a multiplier for how
    much of the mouse input applies to the rotation (in degrees).")]
    public float rotationSpeed = 2.5f;
```

```

[Tooltip("The amount of smoothing applied to the third-person
camera. A higher value will cause the camera to more gradually turn
when the mouse is moved.")]
[Range(0,.99f)]
public float thirdPersonSmoothing = .25f;

[Tooltip("The third-person camera will be kept this many units off of
walls it touches. Setting this higher can help prevent the camera from
clipping with bumpy walls.")]
public float wallMargin = .5f;

[Tooltip("The amount of smoothing applied to the first-person
camera. A higher value will cause the camera to more gradually turn
when the mouse is moved.")]
[Range(0,.99f)]
public float firstPersonSmoothing = .8f;

[Tooltip("Position, local to the Model Holder, for the camera to use
when in first-person mode.")]
public Vector3 firstPersonLocalPosition = new Vector3(0,5.4f,0);

[Tooltip("Position, local to the Model Holder, for the camera to orbit
around when in third-person mode.")]
public Vector3 thirdPersonLocalOrbitPosition = new Vector3(0,5.4f,0);

//Bounds:
[Header("Bounds")]
[Tooltip("Minimum distance from the for the third person camera to have
from its orbit point.")]
public float minThirdPersonDistance = 5;

[Tooltip("Maximum distance from the for the third person camera to have
from its orbit point.")]
public float maxThirdPersonDistance = 42;

[Tooltip("Resembles the current third person distance. Set this to
whatever you want the initial distance value to be.")]
public float thirdPersonDistance = 28;

```

```

[Tooltip("Multiplier for scroll wheel movement. A higher value will
result in a greater change in third-person distance when scrolling the
mouse wheel.")]
public float scrollSensitivity = 8;

[Tooltip("X euler angles for the camera when it is looking as far down
as it can.")]
public int xLookingDown = 65;

[Tooltip("Y euler angles for the camera when it is looking as far up as
it can.")]
public int xLookingUp = 310;

[Header("Misc")]
[Tooltip("The layer mask for what the third-person camera will be
obstructed by, and what it will ignore and pass through. You'll
probably want this to include environmental objects, but not smaller
entities.")]
public LayerMask thirdPersonRayLayermask;

[Tooltip("The key to press to change from first-person to third-person,
or vice versa.")]
public KeyCode modeToggleHotkey = KeyCode.C;

[Tooltip("The key to hold down to hold the camera still, unlock the
mouse cursor, and allow mouse movement.")]
public KeyCode mouseCursorShowHotkey = KeyCode.V;

[Tooltip("Is the camera currently in first-person mode (true) or third-
person mode (false)? This can be set to determine the default mode
when the game starts.")]
public bool firstPerson = true;

//Is the mouse cursor currently showing? Toggled on by holding the
mouseCursorShowHotkey.
private bool showingMouseCursor = false;

//Target position for the third-person camera.
private Vector3 thirdPersonTargetPosition;

```

```

//Gets the third-person camera orbit point in world space.
private Vector3 OrbitPoint
{
    get
    {
        return modelHolder.TransformPoint(thirdPersonLocalOrbitPosition);
    }
}

//Gets the rotation of both the X and Y Camera Targets together.
private Quaternion TargetRotation
{
    get
    {
        //Construct a new rotation out of Euler angles, using the
        //rotation of the X target and Y target together:
        return Quaternion.Euler(cameraXTarget.
            eulerAngles.x,cameraYTarget.eulerAngles.y,0);
    }
}

//Gets a direction pointing forward along the TargetRotation.
private Vector3 TargetForwardDirection
{
    get
    {
        //Return the forward axis of the TargetRotation:
        return TargetRotation * Vector3.forward;
    }
}

//Unity events:
void Start()
{
}

```

```
void Update()  
{  
  
}  
  
void LateUpdate()  
{  
  
}  
}
```

Our tooltips and comments explain the variables and their purposes, so I won't go over it all again. If any of the variables are confusing to you, they'll be explained once they come into play in the following code.

One thing you probably don't recognize is the `LateUpdate` event. This is just like `Update`, in that it is called every frame – however, all scripts will first call their `Update`, and then all scripts will call their `LateUpdate`. This way, if you want to be sure something happens after the rest of your scripts occur, you can place it in `LateUpdate` instead, because by the time `LateUpdates` are being called, every script will have already run its `Update`.

We do this with our camera logic so that any movement the player takes will happen first, before we, for example, raycast using a position relative to our `Model Holder` to determine our target position. This just ensures the camera operations occur after the player movement. If instead the player always moved after the camera determined its target position, then the target position would be “one frame behind” the player movement, which could be noticeable if the framerate drops low enough.

Now, let's get the script ready to run when the time comes. Add the `PlayerCamera` script component to the `Player Camera GameObject` and set the references so that it looks like [Figure 34-1](#). Also make sure to set the `Third Person Ray Layermask` to only include the `Default` layer. If you don't set it, it will include no layers at all, so the third-person camera will never slide against walls like we want it to.

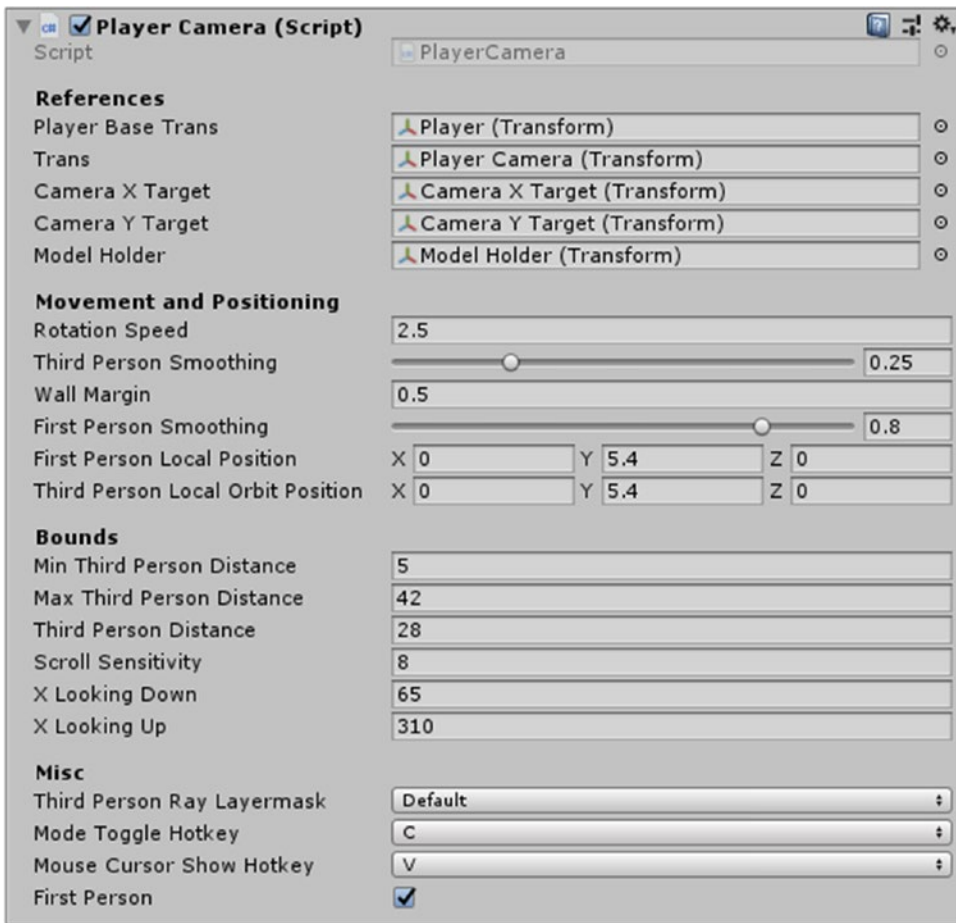


Figure 34-1. View of the Player Camera script component in the Inspector after we've set it up

With that done, let's do our usual routine and map out the basic functionality with some private methods that are called in our Unity event methods:

```
void Start()
{
    //By default, don't show the mouse:
    SetMouseShowing(false);
}

void Update()
{
```

```

//Process hotkeys:
Hotkeys();
}

//LateUpdate occurs after all Update calls for this script and all others.
void LateUpdate()
{
    //Update camera target rotation, so long as we're not showing the mouse
cursor:
    if (!showingMouseCursor)
        UpdateTargetRotation();

    //Perform positioning logic based on the mode we're in:
    if (firstPerson)
        FirstPerson();
    else
        ThirdPerson();
}

```

This outlines the general behavior of our camera. First, the Start method will call a method that we'll declare in a bit: SetMouseShowing. If “false” is passed as a parameter, this will hide the mouse cursor and allow mouse movement to move the camera. If “true” is passed instead, the camera will not respond to mouse movement, and the mouse cursor will show. We'll make use of it in a moment.

The Update method calls only one method: Hotkeys. This is where the pressing of our mode toggling hotkey and our mouse showing hotkey is handled.

The LateUpdate method will call UpdateTargetRotation to handle mouse input and update the rotation of the Camera Targets based on that, but only while the cursor is showing (in other words, while the mouse showing hotkey is not held down). It will then call either the FirstPerson or the ThirdPerson method based on what mode we are in.

Hotkeys

Let's declare our Hotkeys method, **somewhere up above the Unity event methods we just declared**:

```
void Hotkeys()
{
    //Toggling first and third-person mode:
    if (Input.GetKeyDown(modeToggleHotkey))
    {
        firstPerson = !firstPerson;
    }

    //Toggling mouse mode:
    if (Input.GetKeyDown(mouseCursorShowHotkey))
    //Whenever the mouse cursor hotkey is pressed
        SetMouseShowing(true); //Show the mouse

    if (Input.GetKeyUp(mouseCursorShowHotkey))
    //Whenever the mouse cursor hotkey is let go of
        SetMouseShowing(false); //Don't show the mouse
}
```

Whenever the modeToggleHotkey is first pressed down, we use “firstPerson = !firstPerson;” to flip the value of the bool. You'll recall that the “!” can be placed before a value resulting in a bool to “invert” the result, turning true to false or false to true.

After that, the camera will automatically Lerp itself to the position it should be at, which will be implemented in the FirstPerson and ThirdPerson methods, so we don't have to do anything else to make the mode transition.

We also ensure that whenever the mouseCursorShowHotkey is first pressed down, we SetMouseShowing to true, so that the cursor shows and the camera stops responding to mouse movement; when the hotkey is released, we undo that, hiding the cursor again and relieving control to the camera.

The next step would be to actually declare that `SetMouseShowing` method. Let's put it **right above the Hotkeys method**:

```
void SetMouseShowing(bool value)
{
    //Enable or disable the cursor visibility:
    Cursor.visible = value;
    showingMouseCursor = value;

    //Set the cursor lock state based on 'value':
    if (value)
        Cursor.lockState = CursorLockMode.None;
    else
        Cursor.lockState = CursorLockMode.Locked;
}
```

`Cursor.visible` is a built-in static member we can set to hide (false) or show (true) the cursor. But if we just set that, the hidden cursor would still hit the edge of the screen and stop going if the player moved it too far in one direction, and if the game was running in windowed mode, the cursor would begin showing again once the mouse left the screen.

We use `Cursor.lockState` to remedy this. A value of `CursorLockMode.Locked` will lock the cursor at the center of the screen. Even though you can't see it, it will be kept there, which lets us move the mouse as much as we want without it hitting the edges of the game window.

The reason we want a way to easily toggle that off with the `mouseCursorShowHotkey` is so that we have a way to click somewhere else in the Unity editor while testing, such as to change a value in the Inspector or click the Play button to stop playing.

Mouse Input

Let's code our `UpdateTargetRotation` method to handle the Camera Targets. We'll declare this **underneath the Hotkeys method**:

```
void UpdateTargetRotation()
{
    //The X rotation we should be receiving uses the mouse Y because
    rotating along the X axis makes us look up/down.
```

```

float xRotation = (Input.GetAxis("Mouse Y") * -rotationSpeed);

//The Y rotation we should be receiving uses the mouse X because the
rotating along the Y axis makes us look left/right.
float yRotation = (Input.GetAxis("Mouse X") * rotationSpeed);

//Apply the rotation to the camera:
cameraXTarget.Rotate(xRotation,0,0);
cameraYTarget.Rotate(0,yRotation,0);

//We'll keep the camera target's X eulerAngles between the xLookingUp
and xLookingDown values.
//This prevents the camera from looking too far up or down.
if (cameraXTarget.localEulerAngles.x >= 180)
//If the X rotation is anywhere between 180 and 360
{
    if (cameraXTarget.localEulerAngles.x < xLookingUp)
        //and it's less than the looking up value,
        // set it to the xLookingUp:
        cameraXTarget.localEulerAngles = new Vector3(xLookingUp,0,0);
}
else //If the X rotation is anywhere between 0 and 180
{
    if (cameraXTarget.localEulerAngles.x > xLookingDown)
        //and it's past the x looking down value,
        // set it to the xLookingDown:
        cameraXTarget.localEulerAngles = new Vector3(xLookingDown,0,0);
}
}

```

First, we create both an `xRotation` and a `yRotation` variable. The “x” and “y” in the names refer to the axis of rotation that the Camera Target will be receiving. This doesn’t correlate to the X and Y axes of the mouse movement, though, so it might look like we have it backward at first:

- **X rotation** tilts the Transform up toward the sky (negative) or down toward the ground (positive).

Mouse Y input is up (positive) and down (negative). Thus, the Y input is used for the X rotation, but it has to be flipped so that up is negative and down is positive.

- **Y rotation** turns the Transform to the right (positive) or left (negative).

Mouse X input is left (negative) and right (positive). This maps nicely to the Y rotation as is, so we don't have to flip it.

We also multiply the input by `rotationSpeed`, which can be raised or lowered to change the mouse sensitivity.

The rotation is then applied to the corresponding Camera Target Transform: X rotation to the X target and Y rotation to the Y target. Each axis of rotation is performed on a separate Transform to avoid any twisting that would come with applying both rotations to the same Transform.

Afterward, we make sure the X rotation value of the X Target is clamped between the two variables we declared before: `xLookingDown` and `xLookingUp`. The “`localEulerAngles`” member is the “rotation” `Vector3` you'll see in the Inspector when you view a Transform. An X rotation of 0 points the camera directly forward, which is equivalent to 360, where it resets to 0 again. The value can also be depicted as a negative amount, rotating in the opposite direction: a rotation of -10 is equivalent to 350, for example, because a rotation that's less than 0 (a negative rotation) will double back to 360 and decrease from there. Thus, you can depict the same rotation as a negative or positive value: -90 is equivalent to 270, for example.

These two variables define the X rotation values we want to prevent the Target from going past. If looking down as far as possible, it should not be able to tilt any further down than an X rotation of “`xLookingDown`”. Looking up as far as possible, it should not be able to look any further up than an X rotation of “`xLookingUp`”. Figure 34-2 shows the camera on the player, viewed from the right side, looking down at the maximum amount and looking up at the maximum amount. The blue arrow for the Z axis is what we're paying attention to here, since that's the local forward direction of the camera.

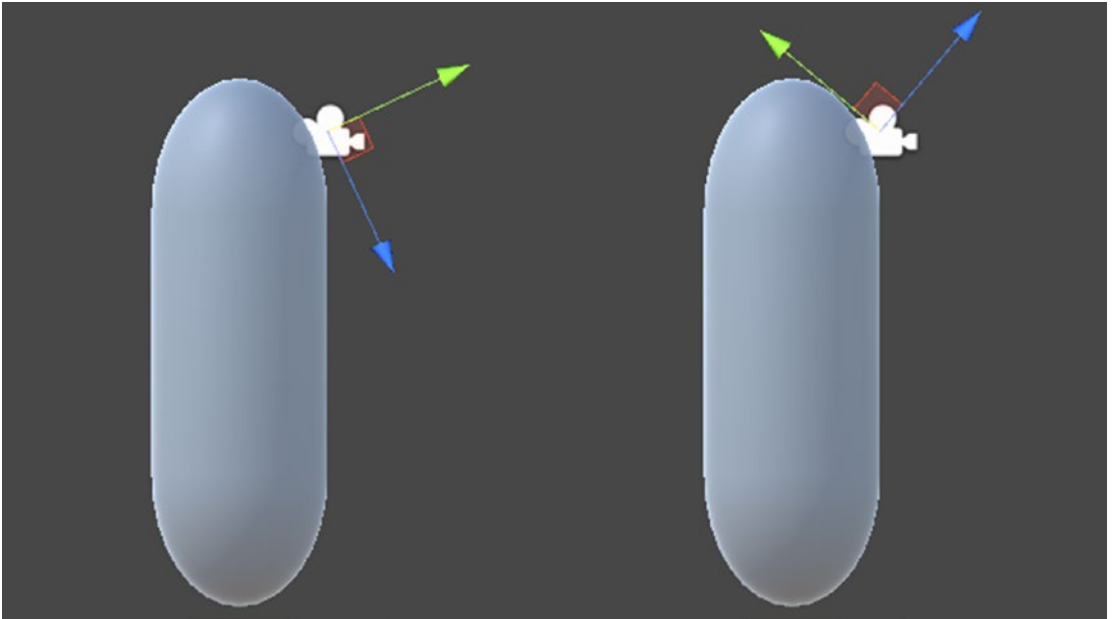


Figure 34-2. Viewing the player and camera from the right side shows the camera looking all the way down (left) and all the way up (right)

Figure 34-3 provides a visualization of the angles involved when viewing the player from the same right-side angle. A rotation of 0, which is equivalent to 360, will point the camera straight forward. Adding to the angle will rotate clockwise, pointing the camera further and further down, while subtracting from it will point it further upward. If it rotates lower than 0, it resets to 360, and likewise, if it rotates higher than 360, it resets to 0. Figure 34-3 also depicts where our `xLookingDown` and `xLookingUp` angles lie.

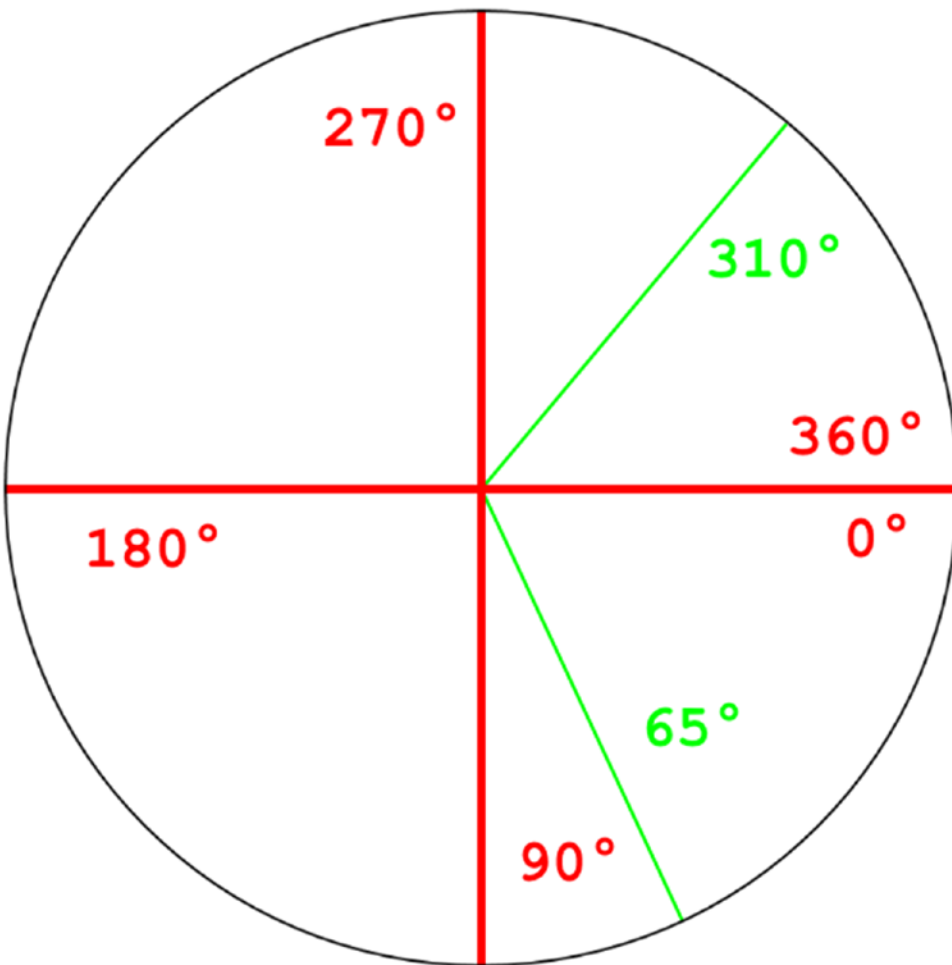


Figure 34-3. Visualization of a single axis of rotation, separated into increments of 90 degrees (red). In green, angles of 65 degrees and 310 degrees are also shown to visualize our default `xLookingDown` and `xLookingUp` values

This should help make sense of why we have to be more particular when we clamp the rotation. Just using a clamp method to clamp the rotation between 65 and 310, for example, would do precisely what we don't want to do. Thus, we separate the logic into two conditions. If the rotation is on the bottom half of that circle – between 0 and 180 – we don't let it raise past `xLookingDown`, which is set to 65. If the rotation is on the upper half of that circle – between 180 and 360 – we don't let it go below the `xLookingUp` value, which is set to 310.

This accounts for the “doubling back” that occurs when the camera goes from, say, 5 degrees to 355 degrees.

First-Person Mode

With the Camera Targets properly rotating, we can implement the first-person camera mode. This will do two things: Lerp the camera toward the `firstPersonLocalPosition` if it is not there already, and then update the camera rotation by Slerping it toward the current rotation of the Camera Target, which is acquired using `TargetRotation` (combining the X Target and Y Target rotations into one Quaternion). We calculate the Slerp result separately and then use only the X and Y Euler angles of that result, to make sure no odd Z rotation occurs. We don't want Z rotation because it would tilt the camera to the left or right, as if the player were cocking their head (imagine the camera doing a barrel roll; that's Z rotation):

```
void FirstPerson()
{
    //If the camera isn't in its first-person location, move it there:
    Vector3 targetWorldPosition = modelHolder.TransformPoint(firstPersonLocalPosition);

    if (trans.position != targetWorldPosition)
    //If the camera isn't at the first person camera location yet
    {
        //Lerp the camera transform towards the first person camera location:
        trans.position = Vector3.Lerp(trans.position, targetWorldPosition, .2f);
    }

    //Get the rotation of the camera, slerped towards the target rotation:
    Quaternion targetRotation = Quaternion.Slerp(trans.rotation,
    TargetRotation, 1.0f - firstPersonSmoothing);

    //Apply just the X and Y axes to the camera:
    trans.eulerAngles = new Vector3(targetRotation.
    eulerAngles.x, targetRotation.eulerAngles.y, 0);

    //Make the model face the same direction way as the camera, but with
    the Y axis removed from the direction:
    modelHolder.forward = new Vector3(trans.forward.x, 0, trans.forward.z);
}
```

First, the camera is moved toward the position we want it to be at while in first-person mode, if it's not already there. This is how we smoothly transition the camera from third-person mode to first-person mode, moving it from its location behind the player to the local position it's expected to have when in first-person mode. To do this, we use the `TransformPoint` method to make the position relative to the `modelHolder` position, rotation, and scale. For example, if a value of `(0, 5.4, 0)` is passed into the `TransformPoint` call, it goes from "5.4 units above the world origin" to "5.4 units above the `modelHolder`."

This system works fine for our purposes, since we'll have our `firstPersonLocalPosition` variable set to `(0, 5.4, 0)`. If we had an X or Z value that was not 0, this could cause the first-person camera to lag slightly behind where it should be when the player is moving their mouse. This is because the model holder is spinning on the Y axis (left and right) as the mouse moves left and right. That doesn't affect the result of transforming the Y position from local to world space, but it will affect the X and Z positions. Thus, the target position changes as the model turns when there is a non-zero value in the X or Z, so it will constantly be smoothing into that position.

Like I said, that won't be a problem for us, but if we did have some model with a head positioned forward or to the side and we wanted to line our first-person camera up with it properly, it could make for an awkward experience.

A workaround to that would be to use a "transitioning" bool variable, marking it as true when we begin transitioning from third- to first-person. While transitioning, use the `Lerp` to move the camera to the target position, and check if it's reached the position yet (or come within a short distance of it). Once it has reached the point, mark "transitioning" as false. While transitioning is false, the camera should be locked in place, so just set its position without using `Lerp`. This keeps the smooth transition when first switching to first-person mode, but after the transition completes, it locks the camera in that place so it doesn't lag behind the point it should be when the model spins.

Third-Person Mode

Our third-person logic will use the `OrbitPoint` that we declared earlier to define the origin of a ray and the `TargetRotation` to determine the direction the ray travels in. Since the `TargetRotation` is always pointing in the direction the camera should be looking forward at, we'll point the ray directly backward from that rotation, traveling only as far

as the `thirdPersonDistance` variable. If it hits a wall, we'll set the target position to that point. Otherwise, we set the target position to the end of the ray.

Of course, we also need to smooth the camera position toward the target position with a Lerp afterward:

```
void ThirdPerson()
{
    //We'll calculate the third-person target position by casting a ray
    backwards from the orbit point.
    //Make a new ray at the position of the orbit point, pointing directly
    backwards from the camera target:
    Ray ray = new Ray(OrbitPoint, -TargetForwardDirection);
    RaycastHit hit;

    //Cast the ray using thirdPersonDistance plus the wall margin to
    account for walls just outside the distance:
    if (Physics.Raycast(ray,out hit,thirdPersonDistance + wallMargin,thirdP
    ersonRayLayermask.value))
    {
        //If the ray hits something, set the target position to the hit point:
        thirdPersonTargetPosition = hit.point;

        //We'll offset it back towards the cameraTarget by 'wallMargin' distance:
        thirdPersonTargetPosition += (TargetForwardDirection * wallMargin);
    }
    else //If the ray didn't hit anything
    {
        //Set the target position to 'distance' units directly behind the
        camera target
        thirdPersonTargetPosition = OrbitPoint - (TargetForwardDirection *
        thirdPersonDistance);
    }

    //Lerp the camera towards the target position using our smoothing
    settings:
    trans.position = Vector3.Lerp(trans.position,thirdPersonTargetPosition,
    1.0f - thirdPersonSmoothing);
}
```

```

    //Now that the camera has been moved properly, make it look at the
    orbit point:
    trans.forward = (OrbitPoint - trans.position).normalized;

    //Make the model face the same direction as the camera, with no Y axis
    position influence:
    modelHolder.forward = new Vector3(trans.forward.x,0,trans.forward.z);
}

```

We declared the `OrbitPoint` property with the rest of the variables. If you check what the `OrbitPoint` actually does, it's just calling `TransformPoint` with the `modelHolder` to get the `thirdPersonLocalOrbitPosition` as a world position, relative to the model holder, much like we did with the first-person local position earlier. Rather than typing out that bulky line each time, we give ourselves a more concise name: `OrbitPoint`.

We also use the `TargetForwardDirection` property we declared before, which uses a simple trick of creating a Quaternion (rotation) out of the combined Euler angles of our X target and our Y target and then multiplying that by a direction: `forward`. This results in a direction that travels forward along the rotation. The property pretty much points in the direction the first-person camera should be pointing, and the opposite direction of this is where the third-person camera aims to be.

You probably recognize the `raycast` method call from our second project. The ray and the “out hit” are passed in, which we just created beforehand. The third parameter is the ray distance, and the fourth is the layer mask value to use.

You may be wondering why the ray distance parameter has the “`wallMargin`” added to it. Think about it like this: if we set our camera up to have a wall margin of 2, that means we’re expecting our camera to distance itself 2 units away from walls behind it. However, if the ray was only as long as the “`thirdPersonDistance`”, and we positioned our camera at the end of the ray whenever there was no wall there, what if a wall was a mere 1 unit, or .005 units, away from the end of the ray? Then our camera would be closer than 2 units to the wall. By extending the ray, we ensure that, if a wall is just a little further away than the tip of the ray, it’ll still get detected and the camera will be kept “`wallMargin`” units away from it as a result.

If the ray hits anything, we set the target position to the hit point and then move it away from the wall using the `TargetForwardDirection`, which is the opposite direction that we sent the ray in. Of course, we use `wallMargin` to determine how far from the wall

it should be moved. If `wallMargin` was set to 0, the target position would simply remain exactly at the hit point.

If the ray did not hit anything, we still have to set the target position. We set it to the `OrbitPoint`, traveling backward from the `Camera Target` – the same direction we used to cast the ray – by the desired `thirdPersonDistance`.

After that, we just have to make sure we’re Lerp’ing the camera position to the target position at all times. This uses the same concept we’ve employed several times before: the smoothing value is “flipped” by doing “1 – smoothing”, because if we just used the smoothing variable as is, a higher value would result in less smoothing, not more, which would be somewhat opposite to what you would expect.

We also have to make sure the camera is always pointing at the `OrbitPoint`. Remember, to get a direction to make an object at position “from” look at an object at position “to”, this is the equation:

`(to - from).normalized`

So our equation gets a direction from the camera position to the `OrbitPoint` and makes that the “forward” direction of the camera.

We also have to spin the model holder to always point forward with the camera. But we can’t allow the camera’s X rotation to influence it. We only want the model holder to spin left and right, not tilt up and down. To do this, we simply construct a new `Vector3` and “cut out” the Y direction of the camera’s forward facing. This gives us a direction that mimics the camera forward facing, but only on the X and Z axes, preventing the possibility of any upward or downward tilting.

Now all that’s left is giving our player a means of changing the third-person distance, which we can tie to the scroll wheel. Since we already have our `Hotkeys` method handling basic input like this, we can add a little chunk of code that changes the distance variable when the scroll wheel is detected. The new code is in bold text in the following:

```
void Hotkeys()
{
    //Toggling first and third-person mode:
    if (Input.GetKeyDown(modeToggleHotkey))
    {
        firstPerson = !firstPerson;
    }
}
```

```

//Toggling mouse mode:
if (Input.GetKeyDown(mouseCursorShowHotkey))
//Whenever the mouse cursor hotkey is pressed
    SetMouseShowing(true); //Show the mouse

if (Input.GetKeyUp(mouseCursorShowHotkey))
//Whenever the mouse cursor hotkey is let go of
    SetMouseShowing(false); //Don't show the mouse

//Scroll wheel for third-person distance:
if (!firstPerson) //Only check for it while we're in third-person mode
{
    //Get scroll wheel delta this frame:
    float scrollDelta = Input.GetAxis("Mouse ScrollWheel");

    //Subtract delta from thirdPersonDistance, multiplying it by the
scroll sensitivity:
    thirdPersonDistance = Mathf.Clamp(thirdPersonDistance - scrollDelta
    * scrollSensitivity,minThirdPersonDistance,maxThirdPersonDistance);
}
}

```

Testing

You should now be able to playtest your camera and watch it in action. Set up a Plane positioned at (0, 0, 0) with a (100, 1, 100) scale to give you some basis of where the ground is, add a tall Cube next to the player so you have something to test the third-person camera wall detection with, and move your mouse around to see it in action. Remember, our default hotkeys are C to switch from first-person to third-person or vice versa and V to show your cursor again and freeze the camera in place. Once you click away from the Game window (e.g., to the Inspector or Hierarchy), you can let go of V, and the camera will remain frozen until you click back into the Game window and tap V again. You can also play with the smoothing values. If you don't notice the smoothing much, turn it up high in the Inspector, and it will become much more obvious.

Summary

With that, we've implemented a camera that can switch smoothly between first- and third-person modes. Our Camera Targets have the rotation applied to them every frame, and the actual camera `GameObject` uses that rotation to determine first-person rotation and third-person target position. We smooth the first-person camera rotation toward its target rotation every frame. For the third-person camera, it is constantly moved toward its target position, orbiting around the `OrbitPoint`, and kept facing forward at the `OrbitPoint`.

CHAPTER 35

Advanced 3D Movement

In this chapter, we'll be implementing the movement system for our player. This includes using the WASD keys to move, pressing Space to jump, and gravity to make the player fall back to the ground. In a later chapter, we'll also implement a wall jumping mechanic that allows the player to "push off" of a nearby wall to gain extra upward and outward momentum while midair.

How It Works

Let's get an overview of how we expect this system to work before we start poking into the code. This movement system will operate with a few concepts that are similar to that of our player movement in the first project:

- Momentum is gained in the direction held by the WASD keys.
- Ongoing momentum is lost over time when no WASD keys are held.
- Attempting to move against ongoing momentum will provide an increase in momentum gain so the player can more easily reverse the direction they're traveling in.

We'll also use a `CharacterController` to perform our movement, just like in the first project. This time, however, we have to account for that pesky Y axis – the player has to fall when they run off of a ledge, so we'll need to make them accumulate downward momentum when they're midair.

To make things slightly more realistic, we'll keep any ongoing momentum when the player becomes midair, whether by jumping or falling off a ledge. This means that all velocity they have acquired through movement will keep going in the same direction after they jump or step off a ledge. The velocity won't begin to drag out until they land on the ground again. Because this can feel a bit sticky, we'll still allow them to influence their movement with the WASD keys while midair, but we'll use a multiplier for midair

movement so we can decrease how effective it is. We want them to be able to draw back if they've made a poor jump or wiggle toward a wall if they didn't jump straight enough at it, but we don't want them to have the same control that they have on the ground.

This time, we have to make the movement local to our player model. In our previous projects, our movement systems used world space directions because the camera never rotated. This time, the player model will be facing whatever direction the player is looking toward. The WASD keys are no longer going to provide velocity in consistent directions. They'll have to be local to the facing of the player.

We can achieve this in a few different ways. Your first idea might be to simply store movement velocity as local to the player. We could do it much the same way as we did in our first project, maintaining a "movementVelocity" variable where the Z axis corresponds to the W and S keys and the X axis corresponds to the A and D keys. For example, if we are traveling straight forward, we might have a movementVelocity of (0, 0, 26). This vector represents 26 units of forward (Z axis) momentum per second.

Since our CharacterController expects us to move it by giving a world space vector, not local to the Transform facing, this means we would have to make our "movementVelocity" local to the facing of our Model Holder, supplying the result to our CharacterController.Move method.

For example, when our movementVelocity is (0, 0, 26), we see it as "26 units of forward momentum, relative to the direction our Model Holder is facing." But the way we want to look at it is not the way our CharacterController will see it. If we pass it into a CharacterController.Move call, it will see it as "26 units forward in world space," which could also be considered "north." It disregards the direction our Model Holder is facing and creates a very awkward experience for our player.

Thus, we have to make the "movementVelocity" local to our Model Holder, using a method on the Transform. This will handle the conversion from local to world space for us. If we transform "movementVelocity" from local to world space when it is (0, 0, 26), and our Model Holder is actually looking backward (south), the "movementVelocity" becomes (0, 0, -26), which takes us south instead of north. This goes for any direction the Model Holder faces. If we are facing directly right, it would become (26, 0, 0) instead or (-26, 0, 0) if we are facing directly left or any shade between.

This is viable and doing it this way can work. We can handle the velocity the same way we did in our first project, the only difference being that we must transform it from local to world space before we supply it to our CharacterController.

But maintaining this is not as easy when it comes to things like falling off of ledges and jumping. Remember, with the movement local to our Model Holder facing, we can obtain forward momentum and then easily steer it by turning the mouse. Whatever way we are facing, that is where the velocity takes us. But once the player becomes midair, we want them to no longer be able to turn their camera to adjust their momentum. You can't just do a big leap forward, then spin around midair, and start going backward. That isn't how it works!

So we must add complications. We would have to convert our “movementVelocity” to world space whenever we become midair. This way, it gets locked into the world direction we were facing when we started the jump and is stuck that way. But then what happens when we land? More complications! The world velocity will have to be converted back to local “movementVelocity” once we land, so that we can steer our momentum again.

This all adds a layer of complexity and finicking to the management of our velocity, and it becomes more awkward when you consider how you might have external velocities applied to the player that are not capped by their maximum movement speed, like if something shoved the player or if the player performed a “dash” move like the one we implemented in our first project.

We'll use a different approach to simplify things. Rather than looking at how far we're moving on each individual axis, we'll look at what is known as the **magnitude** of our velocity Vector3. The magnitude of a vector is a math equation you can perform to get what can be seen as the distance that the vector traverses. It is also sometimes referred to as the **length** of the vector. We can get the magnitude of a vector by simply using the “Vector3.magnitude” property. It returns a float. We don't really need to know what the math equation is, because the plain English version is simpler anyway: it returns “how far the vector travels.” If you were to add Vector3 “A” to a Vector3 position “B”, then “B” will be moving “A.magnitude” units of distance from where it was.

The counterpart of the magnitude, so to speak, is the Vector3.normalized member. A **normalized** vector is essentially “converting it to a direction.” More specifically, it is scaling the magnitude of the vector to 1. The “normalized” member is a property that returns a new vector, going in the same direction but with its magnitude scaled down to 1.

We’ve used it before. Most notably, we have learned that “(to – from).normalized” is how we get the direction to point from a position of Vector3 “from” to another position of Vector3 “to”. For example, to get a direction pointing from an enemy to the player, do this:

```
(playerPosition - enemyPosition).normalized
```

A direction is still a Vector3, it’s just that it has a magnitude of 1. Thus, it can be multiplied by a float to scale the magnitude to whatever that float value is. This is why “.normalized” is effectively turning the vector into a direction. Once you’ve normalized a Vector3, you can then multiply it by a float “X” to get a Vector3 that travels X units in that direction.

We’re about to make use of these concepts to code our movement system. Our velocity will be stored in a single vector depicting the world space velocity our player currently has. This is always world space and never gets converted back and forth.

When we check for input of the WASD keys, we don’t manage our velocity in “units per second.” We just get the local direction that the movement keys are held down in. The local direction is (0, 0, 1) if just W is held. It’s (1, 0, -1) if D (right) and S (backward) are held. And so on.

We can then convert that local direction to a world space direction using the TransformDirection method with our Model Holder reference. It takes a direction that is meant to be local to this Transform and returns back that same direction, but conveyed in world space.

Using that direction, we can then apply our velocity gain per second (a float) to our world velocity. To cap the movement at a maximum speed, we’ll use the magnitude of worldVelocity. If the magnitude is equal to our movespeed variable, we are already moving “movespeed” units per second in the direction we are traveling, so we won’t allow our player to go any faster in that direction (you’ll see how that’s done in a bit).

That’s a general overview of how our process will work – and notably, how it differs from our first project. Before you get overwhelmed with all the details, let’s start implementing it all one piece at a time.

Player Script

We already set up our Player GameObject in the previous chapter, so all we have to do to get set up is add a CharacterController component to the root Player GameObject. Set its Center to (0, 3, 0), its Height to 6, and its Radius to 1, which will make it match the Capsule we added to our Model Holder before.

Now, create a Player script in your Scripts folder. Let's get it started by declaring our variables and an outline of the methods that split up our Update logic:

```
public class Player : MonoBehaviour
{
    //Variables
    [Header("References")]
    [Tooltip("Reference to the root Transform, on which the Player script
    is attached.")]
    public Transform trans;

    [Tooltip("Reference to the Model Holder Transform. Movement will be
    local to the facing of this Transform.")]
    public Transform modelHolder;

    [Tooltip("Reference to the CharacterController component.")]
    public CharacterController charController;

    [Header("Gravity")]
    [Tooltip("Maximum downward momentum the player can have due to
    gravity.")]
    public float maxGravity = 92;

    [Tooltip("Time taken for downward velocity to go from 0 to the
    maxGravity.")]
    public float timeToMaxGravity = .6f;

    //Property that gets the downward momentum per second to apply as gravity:
    public float GravityPerSecond
    {
        get
        {
            return maxGravity / timeToMaxGravity;
        }
    }

    //Y velocity is stored in a separate float, apart from the velocity vector:
    private float yVelocity = 0;
}
```

```

[Header("Movement")]
[Tooltip("Maximum ground speed per second with normal movement.")]
public float movespeed = 42;

[Tooltip("Time taken, in seconds, to reach maximum speed from a stand-
still.")]
public float timeToMaxSpeed = .3f;

[Tooltip("Time taken, in seconds, to go from moving at full speed to a
stand-still.")]
public float timeToLoseMaxSpeed = .2f;

[Tooltip("Multiplier for additional velocity gain when moving against
ongoing momentum. For example, 0 means no additional velocity, .5
means 50% extra, etc.")]
public float reverseMomentumMulitplier = .6f;

[Tooltip("Multiplier for velocity influence when moving while
midair. For example, .5 means 50% speed. A value greater than 1 will
make you move faster while midair.")]
public float midairMovementMultiplier = .4f;

[Tooltip("Multiplier for how much velocity is retained after bouncing
off of a wall. For example, 1 is full velocity, .2 is 20%.")]
[Range(0,1)]
public float bounciness = .2f;

//Movement direction, local to the model holder facing:
private Vector3 localMovementDirection = Vector3.zero;

//Current world-space velocity; only the X and Z axes are used:
private Vector3 worldVelocity = Vector3.zero;

//True if we are currently on the ground, false if we are midair:
private bool grounded = false;

//Velocity gained per second. Applies midairMovementMultiplier when we
are not grounded:
public float VelocityGainPerSecond
{

```

```

    get
    {
        if (grounded)
            return movespeed / timeToMaxSpeed;

        //Only use the midairMovementMultiplier if we are not grounded:
        else
            return (movespeed / timeToMaxSpeed) * midairMovementMultiplier;
    }
}

//Velocity lost per second based on movespeed and timeToLoseMaxSpeed:
public float VelocityLossPerSecond
{
    get
    {
        return movespeed / timeToLoseMaxSpeed;
    }
}

[Header("Jumping")]
[Tooltip("Upward velocity provided on jump.")]
public float jumpPower = 76;

//Update Logic:
void Movement()
{
}

void VelocityLoss()
{
}

void Gravity()
{
}

```

```

    void Jumping()
    {

    }

    void ApplyVelocity()
    {

    }

    //Unity Events:
    void Update()
    {
        Movement();

        VelocityLoss();

        Gravity();

        Jumping();

        ApplyVelocity();
    }
}

```

To define gravity, we declare a maximum downward velocity value that can be applied by gravity and the time we want it to take, in seconds, to reach that maximum velocity. Decrease the `timeToMaxGravity` variable, and gravity will take full effect on your player faster. Increase it and the player will be “floatier,” taking longer to begin falling fast after they have jumped or stepped off an edge.

The actual Y velocity is stored as a float, while the X and Z axes of our velocity will be stored in the “worldVelocity” Vector3. This is so we can track the magnitude of our outward velocity for movement, without the Y axis affecting it. It’ll make more sense why we do it this way when we apply our movement to the velocity.

Our movement variables are similar to those of our first project. The “movespeed” is the maximum velocity on the X and Z axes that we want to be able to have just by moving while grounded. External forces might make us move faster than that, but if we’re just

moving around while grounded, we won't be able to pick up any more speed than "movespeed".

We also use "timeToMaxSpeed" and "timeToLoseMaxSpeed" which are used in the VelocityGainPerSecond and VelocityLossPerSecond properties, just like in the movement system of our first project.

The "reverseMomentumMultiplier" is also the same concept we used in our first project, although we'll be implementing it a little differently. It is a multiplier for our movespeed which is added as bonus speed when we are working to move against ongoing velocity – trying to go right when we are already traveling left, for example. The higher you set it, the quicker the player can switch directions.

The "midairMovementMultiplier" is how we prevent the player from gaining as much velocity from midair movement as they would with grounded movement. We apply it when getting the VelocityGainPerSecond property, but only while "grounded" is false (meaning we're midair).

The "bounciness" is a new variable that we use to determine how hard the player bounces off of walls that they hit while midair. We don't want the player to slide against walls that they strike while midair. If they did, then they would keep sliding against a wall; and if they rose up over the wall, they would keep going. Thus, you could jump at a wall, faceplant into it for a second, and then rise up over it and keep going forward as if you didn't just smack into the wall. To avoid that, we'll redirect the player's momentum away whenever they strike a wall. Alternatively, you could set "bounciness" to 0 to make the player completely stop traveling outward when they hit a wall – they won't bounce off of it, they'll just plop against it like a ball of wet paper towel.

Beneath the variables, our process is outlined with our empty methods:

- **Movement()** will check if the player is holding the WASD keys, updating the "localMovementDirection" vector variable we declared based on which keys are held. If any keys are held, it will convert the local movement direction to world space based on the direction the Model Holder is facing and then apply VelocityGainPerSecond in that direction. Since this occurs first, other methods can use the "localMovementDirection" to check which movement keys are held, if they need to.

- **VelocityLoss()** causes our ongoing velocity to “drag out” while we are grounded; and we are either not holding any movement keys, or our velocity magnitude is greater than “movespeed”.
- **Gravity()** subtracts from “yVelocity” as long as we are midair, but only if our downward velocity has not exceeded “maxGravity”.
- **Jumping()** checks for the Space key being pressed while grounded. If so, it adds “jumpPower” upward momentum by adding to “yVelocity”.
- **ApplyVelocity()** puts our “worldVelocity” and “yVelocity” together and applies it as movement per second with the CharacterController. We’ll use some information given to us by the CharacterController to determine if we touched ground during that movement, and if so, we’ll set “grounded” to true, or otherwise we’ll set it to “false.” Conversely, we’ll also check if we bumped our head during the movement. If so, we’ll lose our upward velocity so we start falling as soon as we hit our head instead of sliding against the surface until gravity begins to pull us down.

To make sure things are ready when we begin, go ahead and add an instance of the Player script to the root “Player” GameObject. Set the three references: “trans” should point to the root “Player” Transform, the Model Holder can be dragged from the Hierarchy onto the corresponding field to reference it, and the CharacterController you just added to the Player can be found and dragged to the Char Controller field through the same Inspector view.

Movement Velocity

Let’s start with basic grounded movement and work up from there. First, we’ll fill in the code for our Movement method, which will make our WASD keys affect our “worldVelocity”. Add the following code to your empty Movement method:

```
void Movement()
{
    //Every frame, we'll reset local movement direction to zero and set its
    X and Z based on WASD keys:
    localMovementDirection = Vector3.zero;
```

```

//Right and left (D and A):
if (Input.GetKey(KeyCode.D))
    localMovementDirection.x = 1;

else if (Input.GetKey(KeyCode.A))
    localMovementDirection.x = -1;

//Forward and back (W and S):
if (Input.GetKey(KeyCode.W))
    localMovementDirection.z = 1;

else if (Input.GetKey(KeyCode.S))
    localMovementDirection.z = -1;

//If any of the movement keys are held this frame:
if (localMovementDirection != Vector3.zero)
{
    //Convert local movement direction to world direction, relative to
    the model holder:
    Vector3 worldMovementDirection = modelHolder.TransformDirection
    (localMovementDirection.normalized);

    //We'll calculate a multiplier to add the reverse momentum
    multiplier based on the direction we're trying to move.
    float multiplier = 1;

    //Dot product will be 1 if moving directly towards existing velocity,
    // 0 if moving perpendicular to existing velocity,
    // and -1 if moving directly away from existing velocity.
    float dot = Vector3.Dot(worldMovementDirection.
    normalized,worldVelocity.normalized);

    //If we're moving away from the velocity by any amount,
    if (dot < 0)
        //Now, flipping the 'dot' with a '-' makes it between 0 and 1.
        //Exactly 1 means moving directly away from existing momentum.
        //Thus, we'll get the full 'reverseMomentumMultiplier' only
        when it's 1.
        multiplier += -dot * reverseMomentumMulitplier;
}

```



```

//Calculate the new velocity by adding movement velocity to the
current velocity:
Vector3 newVelocity = worldVelocity + worldMovementDirection *
VelocityGainPerSecond * multiplier * Time.deltaTime;

//If world velocity is already moving more than 'movespeed' per second:
if (worldVelocity.magnitude > movespeed)
    //Clamp the magnitude at that of our world velocity:
    worldVelocity = Vector3.ClampMagnitude(newVelocity,worldVelocity.magnitude);

//If we aren't moving over 'movespeed' units per second yet,
else
    //Clamp the magnitude at a maximum of 'movespeed':
    worldVelocity = Vector3.ClampMagnitude(newVelocity,movespeed);
}
}

```

As we previously discussed, we'll use the WASD keys to get a "local movement direction" that simply points in the direction the player is holding with the WASD keys. The X and Z axes are all we're using, and they'll either be 0, 1, or -1. This is the direction the player is attempting to move in, local to the facing of the Model Holder.

We convert this from local to world space by calling "modelHolder.TransformDirection", storing the result in the variable named "worldMovementDirection". With this, we know the direction we want our velocity to be influenced toward by our movement, and it's in world space so we can use it to add to our "worldVelocity".

You might wonder why we normalize our "localMovementDirection" when we pass it into the TransformDirection method. Technically, the magnitude is not 1 for our world direction vector if two movement keys are being held. For example, the magnitude of a vector like (1, 0, 1) is not 1, it's a little higher because it traverses a little more distance than a vector that's just (0, 0, 1) or (1, 0, 0). Thus, we actually get a little more movement when we move diagonally, unless we normalize it. This just makes it so that diagonal movement is not "more effective" than moving directly forward, backward, left, or right.

Before we apply the change in velocity using that world direction, we calculate the "multiplier" variable, which is how we apply the reverse momentum influence. This uses a new Vector3 method, "Dot". It returns what is known as the "dot product" of two

Vectors. It should be given two normalized vectors – which means two vectors with a magnitude of 1. As the comments describe, the dot product will be 1 if vector A points in the same direction as vector B, 0 if it points perpendicular (a 90-degree angle away), and -1 if it points in the exact opposite direction. It's not just one of those three values, though – it's a fraction anywhere between them. So if A points in almost the same direction as B, but not exactly, it might return something a little lower than 1, like .9.

We'll use the "dot" to determine how much of our "reverseMomentumMultiplier" gets added to the "multiplier" we declared. First, we check if "dot" is less than 0. If it's greater than 0, it's traveling in a direction no more than 90 degrees off of the direction the world velocity is taking us. Thus, it's not really reversing momentum, so we don't apply any extra multiplier. Since we declare "multiplier" with a default value of 1, this means it's not going to affect the movement at all.

However, if it's less than 0, we add to the multiplier, using "dot" as a fraction for how much of the reverseMomentumMultiplier is used. We flip "dot" so that it's anywhere between 0 and 1, not -1 and 0. If we don't do that, it would decrease the value of "multiplier," since we'd be adding a negative value. Of course, you could also just subtract the value without flipping "dot" if you changed the line to this instead:

```
multiplier -= dot * reverseMomentumMulitplier;
```

Both versions do the same thing in slightly different ways. Use whichever makes more sense to you, if you want!

With that, we have our multiplier. It will be anywhere from 1 to 1.6, if "reverseMomentumMultiplier" is at its default value of .6.

After that, we perform a little vector trickery to apply the velocity. We first calculate the new velocity in a separate vector. This is done by starting with the existing worldVelocity and adding the velocity we want to add on this frame. This equation is simple enough. We use the world movement direction we calculated earlier and multiply that by the velocity we want to gain per second; plus, we apply the "multiplier" we just calculated, and of course, Time.deltaTime is part of the equation, since it is "velocity gained per second."

When we apply the new velocity, we have to make sure we aren't increasing the velocity above the magnitude it should be allowed to have. Since we aren't handling each individual axis (X, Y, and Z) separately as we were in our first project, we have to do this differently. The simple solution is to use the Vector3.ClampMagnitude static method. It takes a Vector3 and a float for the maximum magnitude we want that vector

to be allowed. It returns back the same vector; but, if the magnitude was greater than the float value, it will be scaled down to the float value. If the magnitude was not greater, the vector is returned as is.

We clamp the magnitude in two different ways. If it's already at something greater than "movespeed", then that means some external force may have given us a shove. We don't want to constantly clamp our magnitude at "movespeed" because then this isn't possible anymore. External forces which push us harder than we are able to move on our own will immediately be negated if we constantly clamp our world velocity to "movespeed" magnitude.

But if we aren't moving any faster than "movespeed", we clamp it to a maximum of "movespeed".

This allows us to apply the velocity so that our momentum is adjusted in the direction our movement takes us, but without ever allowing our ongoing momentum to be greater than "movespeed". The same concept applies when the magnitude is greater than "movespeed". Say we're shoved by something, like an enemy striking us or a force field pushing us. We have 60 movespeed, but our world velocity magnitude is now 90 due to the external force. If we move against the momentum, then the clamping of the magnitude doesn't matter. We'll be decreasing the world velocity magnitude because we are losing velocity: moving directly against it, we're simply decreasing its magnitude. But if we move in the same direction, our movement won't give us more speed, since the magnitude is prevented from raising over its current value.

Applying Movement

Let's apply the movement to our player so we can see it in action. We'll add this code to the last method we call in Update, the ApplyVelocity method:

```
void ApplyVelocity()
{
    //While grounded, apply slight downward velocity to keep our grounded
    state correct:
    if (grounded)
        yVelocity = -1;

    //Calculate the movement we'll receive this frame:
```

```

Vector3 movementThisFrame = (worldVelocity + (Vector3.up * yVelocity))
* Time.deltaTime;

//Calculate where we expect to be after moving if we don't hit anything:
Vector3 predictedPosition = trans.position + movementThisFrame;

//Only call Move if we have a minimum of .03 velocity:
if (movementThisFrame.magnitude > .03f)
    charController.Move(movementThisFrame);

//Checking grounded state:
if (!grounded && charController.collisionFlags.HasFlag(CollisionFlags.
Below))
    grounded = true;
else if (grounded && !charController.collisionFlags.
HasFlag(CollisionFlags.Below))
    grounded = false;

//Bounce off of walls when we hit our sides while midair:
if (!grounded && charController.collisionFlags.HasFlag(CollisionFlags.Sides))
    worldVelocity = (trans.position - predictedPosition).normalized *
    (worldVelocity.magnitude * bounciness);

//Lose Y velocity if we're going up and collided with something above us:
if (yVelocity > 0 && charController.collisionFlags.
HasFlag(CollisionFlags.Above))
    yVelocity = 0;
}

```

Because we'll be asking our CharacterController "Did we hit something below us the last time we called Move()?" to determine if we are grounded or not, we apply a constant, negligible amount of downward velocity while we are grounded. This way, if we move while grounded, we'll go down a little bit, causing us to touch the floor beneath us. If we didn't do this, we'd move directly outward, and the CharacterController would not think we were grounded because our bottom didn't touch anything.

We store the vector we will be moving on this frame in a variable and later pass that into the Move call with our CharacterController. The velocity is simple enough: worldVelocity is our X and Z velocity, and we add (0, yVelocity, 0) to that. Remember,

`Vector3.up` is just a shorthand way of typing “new `Vector3(0, 1, 0)`”. Multiplying a float by `Vector3.up` is just saying “go up by this amount.”

We also store a vector for the position we expect to have after moving, if nothing gets in our way. This is used to calculate bouncing direction.

To prevent calling `Move` when there’s barely any velocity, we only call it when the distance we are going to move is greater than `.03`. This will help us prevent an issue down the road with platforms. It’ll also cut out `Move` calls that aren’t really moving us anywhere noticeable, which can save a little on performance.

After we move, we can then use the `CharacterController.collisionFlags` member to check which parts of the capsule making up our controller had collisions during the last move call.

This is a bit mask, which behaves like a layer mask. Remember how layer masks are essentially a list of “checkboxes” for each entry? Each individual layer can be true or false. This is how the collision flags work, except instead of layers, we have collision directions: Below, Sides, and Above. We can use the “`HasFlag`” method to return true if a collision occurred Below, at the Sides, or Above. Of course, it will return false if a collision did not occur there.

We check if we are currently not grounded and hit something below us. If so, we become grounded.

After that, we check if we are grounded, but did not hit anything below us. In that case, we must become midair (not grounded).

We also perform our “bouncing” here. When we hit something from our side while midair, we adjust our velocity. This is done by redirecting it from the predicted position to the actual position we ended up at. We then multiply that direction by the magnitude, which is affected by “bounciness.” If the bounciness is 1, we get the full magnitude redirected. If it were `.5` instead, we’d only get 50% of the magnitude, causing some of our momentum to be lost when we hit the wall.

After that, we check also for collisions at our top. If we struck something above us, we lose all positive `yVelocity`. If we didn’t implement this, we would keep rising up against anything above us until gravity dropped our `yVelocity` below 0. This way, it immediately drops to 0 once we bump our head, causing us to start falling.

With that, you can test movement by using the WASD keys. Of course, we still have to actually make the movement stop once we let go of the WASD keys; otherwise, we’ll just keep moving.

Losing Velocity

Let's implement the `VelocityLoss` method:

```
void VelocityLoss()
{
    //Lose velocity as long as we are grounded, and we either are not
    //holding movement keys, or are moving faster than 'movespeed':
    if (grounded && (localMovementDirection == Vector3.zero ||
        worldVelocity.magnitude > movespeed))
    {
        //Calculate velocity we'll be losing this frame:
        float velocityLoss = VelocityLossPerSecond * Time.deltaTime;

        //If we're losing more velocity than the world velocity magnitude:
        if (velocityLoss > worldVelocity.magnitude)
            //Zero out velocity so we're totally still:
            worldVelocity = Vector3.zero;

        //Otherwise if we're losing less velocity:
        else
            //Apply velocity loss in the opposite direction of the world
            //velocity:
            worldVelocity -= worldVelocity.normalized * velocityLoss;
    }
}
```

This isn't terribly complicated. We first supply the situation when velocity loss should occur:

- We must be grounded, not midair.
- We must not be holding any of the WASD keys. To determine this, we use the “`localMovementDirection`” vector which we set in the `Movement` method every frame.
- Alternatively, if we are holding any of the WASD keys, we will still lose velocity if our world velocity magnitude is greater than “`movespeed`”.

To apply the velocity loss, we first calculate how much magnitude we should lose on this frame in a quick variable. Then, we must apply it one of two ways. If the magnitude we are losing on this frame is greater than the magnitude of our world velocity, then applying it should just end all momentum, so we set worldVelocity to “zero.”

Otherwise, if we aren’t going to lose all velocity magnitude on this frame, we apply the velocity loss as momentum in the opposite direction that the worldVelocity is currently traveling in. Seeing this, it should become a bit clearer why we must differentiate between the two methods of applying the velocity. If we just did the latter method every frame, then we would never actually stop moving completely. We would apply velocity in the opposite direction until our momentum reversed; then we’d do it again and again, constantly reversing the direction because we’re constantly adding some amount of velocity every frame.

That should now allow us to move around in-game and, once we let go of the WASD keys, lose all of our velocity over time.

Gravity and Jumping

Now all that’s left is the vertical axis. First, we’ll fill in the Gravity method, which is a simple few lines of code:

```
void Gravity()
{
    //While not grounded,
    if (!grounded && yVelocity > -maxGravity)
        //Decrease Y velocity by GravityPerSecond, but don't go under
        -maxGravity:
        yVelocity = Mathf.Max(yVelocity - GravityPerSecond * Time.
        deltaTime, -maxGravity);
}
```

Since maxGravity is set as a positive value, depicting the “maximum downward momentum we can have due to gravity,” we have to do a little flipping when we apply it to yVelocity. If yVelocity is positive, we’ll go up. If it’s negative, we’ll go down. Thus, the gravity needs to subtract from our yVelocity. We use Max to ensure that should it drop below “-maxGravity”, it instead is set to “-maxGravity”.

We only do this if our `yVelocity` is not already less than “`-maxGravity`”. This makes sure that external forces can drive us downward harder than gravity can, but should that happen, gravity will not keep applying.

With that, you can add a cube to your scene to walk on, position your player on top of it, and then play and walk off the edge. You should start falling as soon as your figurative feet leave the cube.

Let’s give ourselves a way to get back up onto the cube, though, and implement jumping:

```
void Jumping()
{
    if (grounded && Input.GetKeyDown(KeyCode.Space))
    {
        //Start traveling 'jumpPower' upwards per second:
        yVelocity = jumpPower;

        //Stop counting ourselves as grounded since we know we just jumped:
        grounded = false;
    }
}
```

Again, not too complicated. We only allow jumping while grounded, and it occurs when you press Space. Since we know we’re grounded and will have no downward velocity (except the default `-1` to keep ground detection functioning correctly), our `yVelocity` can simply be set to the “`jumpPower`” with an “`=`” rather than adding to it with a “`+=`”.

You might wonder why it’s necessary to bother setting “`grounded`” to false when a jump occurs. You’ll recall that, while grounded, we constantly set our “`yVelocity`” to `-1` in the `ApplyVelocity` method, which is called just after the `Jumping` method. This would still occur immediately after a jump if we didn’t set “`grounded`” to “`false`” here, which would make jumping do nothing.

With that, you can test again and try jumping with Space. You’ll rise and fall based on the gravity settings and the jump power. How high the jump takes you is dependent upon a combination of all those variables: the maximum gravity, the time taken to apply maximum gravity, and the jump power.

Summary

In this chapter, we learned some more advanced tricks for working with vectors to implement player movement, jumping, and gravity in a mouse-aimed setup. Some key things to remember are as follows:

- The **magnitude** (also called **length**) of a vector is the amount of distance it traverses.
- A **normalized** vector is a vector with a magnitude of 1. This can be looked at as a “direction.” Multiply it by float “X” to go X units in the given direction.
- After calling Move with a CharacterController, you can test where collisions occurred on the collider using the “collisionFlags” member.

CHAPTER 36

Wall Jumping

With our player movement, gravity, and jumping implemented in the last chapter, we'll move on to give our player the ability to push off a nearby wall for an extra midair jump.

There are different ways to design a wall jumping mechanic. You might only allow a wall jump to be performed in the opposite direction that the player's velocity is traveling in, to enforce the idea that they are "pushing off" of the wall and redirecting their momentum.

Our system will be a bit more allowing. We'll simply test for collisions with walls near the player, and if we can find any, we'll allow the wall jump. The wall jump will go straight up if the player is holding no WASD movement keys. If they are holding any keys, it will also push off in the local direction they're holding. For example, holding S will attempt to wall jump backward or D to wall jump toward the player's right side.

Variables

First off, let's declare the variables that relate to wall jumping. Everything's going in the Player script, of course, so open it up. Beneath the last variable we declared, which was "jumpPower", add these variables:

```
[Header("Wall Jumping")]
[Tooltip("Outward velocity provided by wall jumping.")]
public float wallJumpPower = 40;

[Tooltip("Upward velocity provided by wall jumping.")]
public float wallJumpAir = 56;

[Tooltip("Maximum distance from the player's side that a wall can be
detected for a wall jump.")]
public float wallDetectionRange = 2.4f;
```

```

[Tooltip("Cooldown time for wall jumps, in seconds.")]
public float wallJumpCooldown = .3f;

[Tooltip("Only layers included in this mask will count as walls that can be
jumped off.")]
public LayerMask wallDetectionLayerMask;

//Time.time when we last performed a wall jump.
private float lastWallJumpTime;

//Returns true if wall jump is not on cooldown, false if it is on cooldown.
private bool WallJumpIsOffCooldown
{
    get
    {
        //Current time must be greater than the last wall jump time, plus
        wall jump cooldown:
        return Time.time > lastWallJumpTime + wallJumpCooldown;
    }
}

```

- **“wallJumpPower”** is the velocity we’ll be applying on the X and Z axes when a wall jump occurs. This is only applied if any of the WASD keys are held when the jump is first ordered.
- **“wallJumpAir”** is the upward velocity, which will always be applied regardless of the WASD keys.
- **“wallDetectionRange”** depicts the maximum distance away from the player a wall can be when we wall jump off it. Anything that’s further from the player than this will not be detected for wall jumping.
- **“wallJumpCooldown”** is a short cooldown we’ll apply to wall jumps so the player can only perform one every .3 seconds.
- **“wallDetectionLayerMask”** will be used when we check for walls near the player.

- **“lastWallJumpTime”** will be set to the current `Time.time` whenever we perform a wall jump. It will be used to check if the wall jump is on cooldown.
- **“WallJumpIsOffCooldown”** is a shorthand property for checking if the wall jump is off cooldown (true) or on cooldown (false).

Once you’ve written these variables, be sure to save the script and go set up that layer mask in the Inspector for your Player. We’ll make it include just the Default layer, not any of the others.

In the Inspector, our new Wall Jumping variables will look like Figure 36-1 when finished.

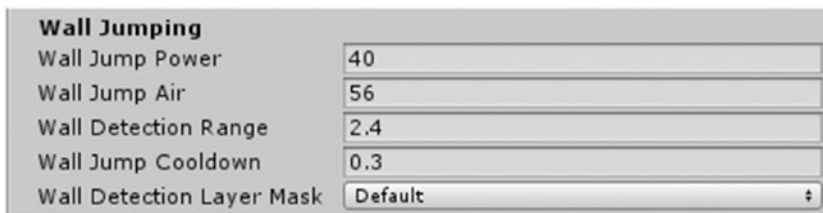


Figure 36-1. The Wall Jumping variables of our Player script in the Inspector

Detecting Walls

We’ll need to know if a wall is near us before we allow a wall jump to occur. To detect if a wall is nearby, we’ll call the method `Physics.OverlapBox`. This takes some parameters defining an invisible box in world space. It tests if any colliders are inside this box or touching it, gathers them all in a `Collider[]` array, and then returns that array.

All we’ll need is to know if at least one collider is returned to us. We don’t really need to mess with the returned array. We just need to check if its `Length` member is greater than 0. To do this, we can define a `WallIsNearby` method that performs the check using `OverlapBox` and then returns true if the returned array length is greater than 0 or false if it is not.

If we’re clever about it, this can be done in a single line of code within the `WallIsNearby` method, but your first instinct is probably to do it in more than one line. I’ll give a few code samples that you don’t have to write into your Player script yet, just to demonstrate the point.

The following code sample shows one way of doing it, with the parameters of the `OverlapBox` call excluded just to keep the extra clutter out of the way for now:

```
private bool WallIsNearby()
{
    Collider[] colliders = Physics.OverlapBox();

    if (colliders.Length > 0)
        return true;
    else
        return false;
}
```

Pretty simple, right? Store the returned array in a local variable and check the `Length` member. It's not necessarily wrong, it's just doing it in more lines of code than necessary.

Here is how you might do the same thing with one line of code in the method instead:

```
private bool WallIsNearby()
{
    return Physics.OverlapBox().Length > 0;
}
```

As you can see, we are calling `OverlapBox`, but we're not storing the returned array in a local variable before we access it. We're simply reaching directly into the returned array after the closing `)` of the `OverlapBox` method call. Doing this, we can grab the `Length` member of the array and use the `>` operator to get a bool value: true if the array has any members in it and false if it does not.

This comes out to one simple equation that can be returned as it is, no variables required. As long as the value coming after the `return` keyword evaluates to the correct type, you can use operators and method calls to get the value. You don't have to just `return false` or `return true`!

With that out of the way, let's write the actual method, including the parameters we'll use to define the `OverlapBox`. To spread the method call out so it's a little more comfortable to read, I'll put each parameter on its own line of code. This is the final version of the method, so declare it **beneath our Wall Jumping variables in the Player script**:

```
private bool WallIsNearby()
{
    return Physics.OverlapBox(
        trans.position + Vector3.up * (charController.height * .5f),
        Vector3.one * wallDetectionRange,
        modelHolder.rotation,
        wallDetectionLayerMask.value).Length > 0;
}
```

This is something you may see when method calls have lots of parameters or when the parameters are large and unwieldy. Each parameter is given with its comma at the end to separate it from the next parameter, but we also add a line break after the comma.

Let's go over what these parameters are.

The first parameter is a `Vector3` depicting the world position of the center of the box. We start at the root `Transform` position, which is at the floor level, right at the bottom of our `Capsule` model. We add a vector going straight up by half of our `CharacterController`'s `Height` setting, which we've set in the `Inspector` to a value of 6. So effectively, the box center is halfway up the height of the player.

The second parameter is the "half extents" of the box. This is a `Vector3` depicting half of the size of the box. `X` is width, `Y` is height, and `Z` is length. It is given as "half extents" instead of the full size of the box.

For this parameter, we use "`Vector3.one`", which is shorthand for "`new Vector3(1, 1, 1)`". We multiply it by the `wallDetectionRange`. This is effectively the same as typing this:

```
new Vector3(wallDetectionRange, wallDetectionRange, wallDetectionRange)
```

It's just a bit shorter. Ultimately, this parameter depicts that each side of the box will be "`wallDetectionRange`" away from the center. The total size of the box is double the `wallDetectionRange`, if you measure it from one side to the opposite.

The third parameter is a `Quaternion` for the rotation of the box. We give the box `Quaternion.identity` for its rotation, which just means "no rotation." It will point forward along the world forward direction, just as if you had created a new `Cube` in the `Scene`.

Lastly, the fourth parameter is our layer mask value, which lets us define which layers constitute as walls that can be jumped off. We've seen this previously in our use of the `raycast` method.

Performing the Jump

With everything set up, let's add an extra method to check for the Space key being pressed and perform the wall jump if applicable.

We'll add a Wall Jumping method amid our existing Update logic methods we've declared in the past, **between Gravity and Jumping**:

```
void Movement() {...}

void VelocityLoss() {...}

void Gravity() {...}

void WallJumping()
{
}

void Jumping() {...}

void ApplyVelocity() {...}
```

And we'll call it in the Update method, of course. We want it to occur before our Jumping method:

```
void Update()
{
    Movement();

    VelocityLoss();

    Gravity();

    WallJumping();

    Jumping();

    ApplyVelocity();
}
```

Now let's fill in the WallJumping method:

```
void WallJumping()
{
    //If midair and wall jump is off cooldown:
    if (!grounded && WallJumpIsOffCooldown)
    {
        //If space is pressed:
        if (Input.GetKeyDown(KeyCode.Space))
        {
            //Make sure a wall is nearby to jump off:
            if (WallIsNearby())
            {
                //If any movement keys are held,
                if (localMovementDirection != Vector3.zero)
                {
                    //Apply outward movement by converting local movement
                    direction to world-space
                    // relative to the model holder, and multiplying by
                    wall jump power:
                    worldVelocity = modelHolder.TransformDirection(local
                    MovementDirection) * wallJumpPower;

                    //We'll also apply Y velocity. If we're falling,
                    if (yVelocity <= 0)
                    {
                        // all downward momentum is replaced with the wall jump air:
                        yVelocity = wallJumpAir;
                    }

                    //If not falling, just add wall jump air to existing velocity:
                    else
                    {
                        yVelocity += wallJumpAir;
                    }

                    //Apply wall jump cooldown:
                    lastWallJumpTime = Time.time;
                }
            }
        }
    }
}
```


The first three if's amount to "while midair and wall jump is off cooldown, when the Space key is first pressed, and if a wall is nearby." On those conditions, we perform the wall jump.

To apply outward momentum, we transform the local movement direction (which WASD keys are held) to world space and multiply that by the wall jump power. We directly set the worldVelocity to this value with an "=" operator rather than adding the velocity as extra. This means if you jump off a wall, all existing outward velocity will be ended, and the wall jump velocity will replace it.

For example, this way the player can jump directly at a wall and wall jump backward in the opposite direction. If we just added the wall jump velocity to the world velocity, then attempting to go backward with a wall jump wouldn't be as effective, since it would be working against ongoing momentum. The two would likely just cancel each other out, unless your wall jump power was sufficiently high.

If you would rather have ongoing momentum be retained when a wall jump occurs, you can change the line to include the current world velocity magnitude as well as the jump power:

```
worldVelocity = modelHolder.TransformDirection(localMovementDirection) *
(wallJumpPower + worldVelocity.magnitude);
```

This is the same thing, but rather than just multiplying by wallJumpPower, we multiply by the power plus the ongoing magnitude of our world velocity. In other words, however fast the velocity was moving before is added to the wall jump power, but it's all directed along the wall jump direction instead of whatever direction we were traveling before.

This can make for more convincing wall jumps if you were, say, pushed by an external force. Rather than all of your momentum being replaced by the wall jump power, which could look awkward if the wall jump power was less than the momentum you had, your existing momentum gets redirected and the wall jump power is also added to it as extra velocity. Wall jumping back and forth between two walls repeatedly this way could "stack up" a lot of velocity, though.

In the end, it's just a matter of how you want to implement the mechanic. For our purposes, I'll be leaving it at the first example, with no "worldVelocity.magnitude" involved. If you think it's more fun the second way, go ahead and replace the line.

After we add that outward velocity, we then add the upward velocity. This is done one of two ways. If we're falling, we want the wall jump to counteract that downward

momentum, so we directly set “yVelocity” rather than adding to it. This overrides any negative velocity we already had.

Otherwise, if we wall jump when we’re already rising, we just add extra upward velocity.

Lastly, we set the Time.time at which the last wall jump was performed, which puts it on cooldown.

With that, you can go and test out the new features. Try creating some cubes and making them tall enough to jump up against and wall jump off. You can stand next to the wall, jump with no WASD keys held, and press Space again while midair to just go straight up. The default cooldown is low enough that you can keep going up and up this way (although you can raise it if that sort of power frightens you).

You can also put two tall cubes next to each other and wall jump back and forth between them.

Remember, if you forgot to set up the layer mask in the Inspector, the walls around the player might not be detected, preventing you from performing wall jumps. Similarly, if you neglected to change the Player’s layer, the player might still be layered “Default” causing them to count as a wall themselves, which would allow wall jumping even if no wall is nearby.

Summary

In this chapter, we learned how to use the Physics.OverlapBox method to test for colliders within a box-shaped area. Using this, we allow the player to press Space while midair with a wall nearby to push off the wall for an extra aerial jump.

CHAPTER 37

Pulling and Pushing

In this chapter, we'll make use of raycasting to allow the player to point their camera at a `GameObject` with a `Rigidbody` and hold left-click to pull the object toward them or right-click to push it away from them. Rather than pulling and pushing by moving the `Transform` directly, we'll apply force to the `Rigidbody` so that the physics system handles the motion instead for us.

Script Setup

The pushing and pulling features will be implemented in a separate “Telekinesis” script that we'll attach to the Player `GameObject`. We'll do this instead of writing everything in the Player script, just to keep things organized.

Start off by creating a script named `Telekinesis` in the `Scripts` folder of your project. Open it up, and let's declare our variables:

```
public class Telekinesis : MonoBehaviour
{
    public enum State
    {
        Idle,
        Pushing,
        Pulling
    }

    private State state = State.Idle;

    [Header("References")]
    public Transform baseTrans;
    public Camera cam;
```

```

[Header("Stats")]
[Tooltip("Force applied when pulling a target.")]
public float pullForce = 60;

[Tooltip("Force applied when pushing a target.")]
public float pushForce = 60;

[Tooltip("Maximum distance from the player that a telekinesis target
can be.")]
public float range = 70;

[Tooltip("Layer mask for objects that can be pulled and pushed.")]
public LayerMask detectionLayerMask;

//Current target of telekinesis, if any.
private Transform target;

//The world position that the target detection ray hit on the current target.
private Vector3 targetHitPoint;

//Rigidbody component of target. For something to be marked as a
target, it must have a Rigidbody.
//So as long as 'target' is not null, this won't be null either.
private Rigidbody targetRigidbody;

//If there is no current target, this is always false. Otherwise, true
if the target is in range, false if they are not.
private bool targetIsOutsideRange = false;

//Gets the Color that the cursor should display based on the state and
target distance.
private Color CursorColor
{
    get
    {
        if (state == State.Idle)
        {
            //If there is no target, return gray:
            if (target == null)

```

```

        return Color.gray;

//If there is a target but it's not in range, return orange:
else if (targetIsOutsideRange)
    return new Color(1,.6f,0);

//If there is a target and it is in range, return white:
else
    return Color.white;
}
//If we're pushing or pulling, return green:
else
    return Color.green;
}
}
}

```

Our variables explain themselves in their tooltips and comments, so let's go over how the system works and where the variables find their purpose.

Every frame, we'll cast a ray using the "detectionLayerMask" with infinite distance (range). If a valid target is found with the ray, we'll set the related variables:

- target
- targetHitPoint
- targetRigidbody
- targetIsOutsideRange

This gives us all we need to know about our target, if we have one. As you can see, we still target objects that are outside of our "range" variable, but we have the "targetIsOutsideRange" bool to tell us if the target can actually be pulled or pushed.

We then check for input: holding the left mouse button while we have a valid, in-range target will pull the target toward us, while right-click will push the target away.

We'll set our "state" based on what we were doing on this frame: nothing (Idle), pushing, or pulling.

Our CursorColor property reacts to the “state” as well as whether we have a target that is in range, returning a different Color based on these factors:

- If there is no target, CursorColor returns **gray**.
- If there is a target but it is outside the range, CursorColor returns **orange**.
- If there is a target and it is inside range, CursorColor returns **white**.
- While we are pushing or pulling a target, CursorColor returns **green**.

The “cursor” is a small dot we’ll draw in the center of the screen, where the raycast originates. Of course, this dot will use CursorColor to define its color. This will make it automatically update based on the situation, to give the player some indication of when they have a valid target, when the target is outside range, and when they’re actively pulling or pushing their target.

Before we continue, let’s set up the Telekinesis script. First, add an instance of the script to your root Player GameObject (the same one that has the Player script instance). Set the “baseTrans” reference to the Player Transform and drag and drop the Player Camera onto the “cam” reference field. Make sure to also set the layer mask to include only the Default layer. When you’re done, your script should look something like Figure 37-1 in the Inspector.

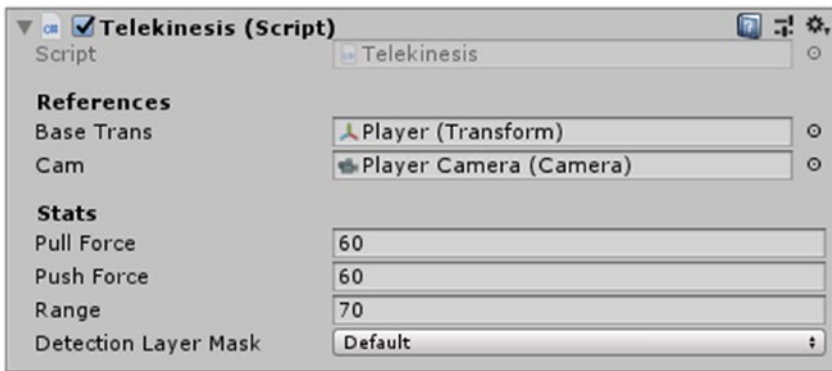


Figure 37-1. Our Telekinesis script with all of its fields correctly set in the Inspector

Moving on, let's map out our basic functionality with some methods:

```
//Update logic:
void TargetDetection()
{
}

//FixedUpdate logic:
void PullingAndPushing()
{
}

//Unity events:
void Update()
{
    TargetDetection();
}

void FixedUpdate()
{
    PullingAndPushing();
}
```

You'll notice we're using a new built-in Unity event here: `FixedUpdate`. This is where the actual pulling and pushing will be performed, while the raycast for target detection will instead occur in the normal `Update` event that we're so used to.

FixedUpdate

The `FixedUpdate` method is like `Update`, but you should use `FixedUpdate` instead if you intend on interacting with the physics system through code. Notably, applying forces to `Rigidbody`s should be done through `FixedUpdate` instead of `Update`. It occurs not once per frame, but at a set interval with the same amount of time between each `FixedUpdate`. Unity warns that unexpected results can occur in physics components if you interact with them in `Update` instead of `FixedUpdate`!

How frequently the physics updates are called is dependent on a value that you can set by navigating to the Edit ► Project Settings window, clicking the Time tab, and locating the Fixed Timestep value, shown in Figure 37-2. By default, the value is set to .02, which means FixedUpdates are called 50 times per second.

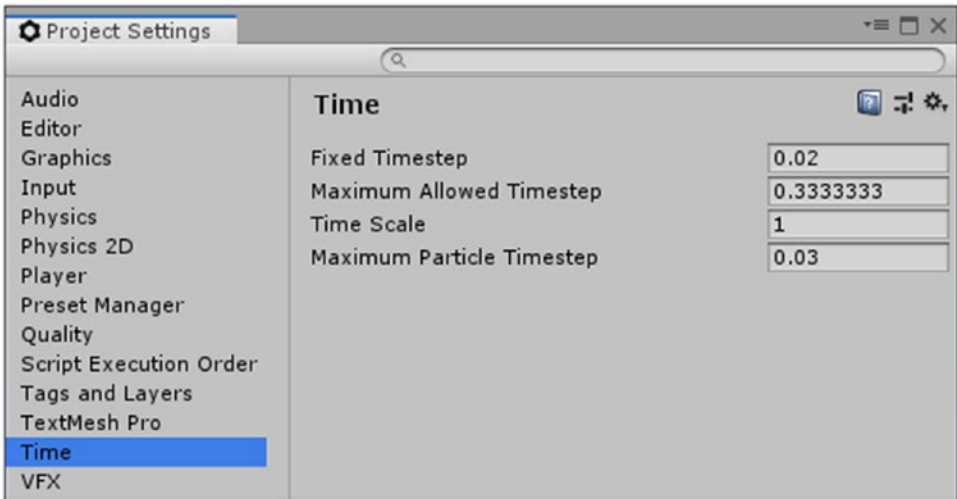


Figure 37-2. The Time tab is open in the Project Settings window, where the Fixed Timestep field is at the top of the listings

Setting the value lower will generate more updates per second but comes at the cost of performance. Of course, setting it higher will save on performance but may make physics less accurate or downright choppy at particularly high values.

Within a FixedUpdate call, Time.deltaTime will still work the same way, returning the Fixed Timestep value always. You can also access Time.fixedDeltaTime to get this value within your code. You can even set it in-game to dynamically change the update frequency of physics.

Target Detection

Before coding our FixedUpdate logic, let's get our target detection working so we know what we're dealing with.

We'll fill in the TargetDetection method we declared before with this code:

```
void TargetDetection()
{
    //Get a ray going out of the center of the screen:
    var ray = cam.ViewportPointToRay(new Vector3(.5f,.5f,0));
    RaycastHit hit;

    //Cast the ray using detectionLayerMask:
    if (Physics.Raycast(ray,out hit,Mathf.Infinity,detectionLayerMask.value))
    {
        //If the ray hit something,
        if (hit.rigidbody != null && !hit.rigidbody.isKinematic) // and it
            has a non-kinematic Rigidbody,
        {
            //Set the telekinesis target:
            target = hit.transform;
            targetRigidbody = hit.rigidbody;
            targetHitPoint = hit.point;

            //Based on distance, set targetIsOutsideRange:
            if (Vector3.Distance(baseTrans.position,hit.point) > range)
                targetIsOutsideRange = true;
            else
                targetIsOutsideRange = false;
        }

        //If the thing the ray hit has no Rigidbody
        else
            ClearTarget();
    }
    else //If the ray didn't hit anything
        ClearTarget();
}
```

Here, we exhibit usage of the `Camera.ViewportPointToRay` method. This method is just like the `ScreenPointToRay` method that we used in our second project to detect where to place our tower building highlighter. The only difference is that it operates by the “viewport” instead of by a pixel position on the screen. It’s just a different way to locate a position on the camera view. Rather than specifying pixels, such as half of the width and height of the screen, we specify a fraction between 0 and 1 for the X and Y values. The X is left and right, and the Y is up and down, just like with pixels, but we don’t have to concern ourselves with the screen width and height. (0, 0) is the bottom-left corner of the camera, and (1, 1) is the top-right corner. Thus, (.5f, .5f) will get us the center. Since we don’t have to plug the mouse position into the method, this one will suit us just fine as an easy way to get a ray shooting out of the center of the screen.

The Z axis doesn’t do anything, so we just leave it at 0.

We cast the ray. If the ray hit anything, the “hit” will be filled with data about what was hit, as we’ve come to understand about raycasting.

We’ll only mark something as a target if it has a `Rigidbody` and only if that `Rigidbody` is not kinematic. You’ll recall that a kinematic `Rigidbody` is not controlled by the physics system. We can’t apply forces to such a `Rigidbody` anyway, so they don’t make for valid targets.

We set our four target-related variables for future reference.

If the target did not have a non-kinematic `Rigidbody`, or if the ray simply didn’t hit anything in the first place, we call a `ClearTarget` method.

Let’s declare that method. It’s a simple one that just resets the values of the variables to null and false. I’ll put it **down below the `CursorColor` property**:

```
void ClearTarget()
{
    //Clear and reset variables that relate to targeting:
    target = null;
    targetRigidbody = null;
    targetIsOutsideRange = false;
}
```

That does it for target detection. We can now expect our camera to constantly be shooting a ray out of its center, striking only the layers defined in our “`detectionLayerMask`”. It will detect and store information about the target the ray strikes, if any. Otherwise, it clears the target.

Pulling and Pushing

Now we can fill in the method that does the interesting part: detecting mouse buttons and applying forces to pull or push the target.

We'll fill in the `PullingAndPushing` method with this code:

```
void PullingAndPushing()
{
    //If we have a target that is within range:
    if (target != null && !targetIsOutsideRange)
    {
        //If the left mouse button is down
        if (Input.GetMouseButton(0))
        {
            //Pull the target from the hit point towards our position:
            targetRigidbody.AddForce((baseTrans.position - targetHitPoint).
            normalized * pullForce, ForceMode.Acceleration);
            state = State.Pulling;
        }

        //Else if the right mouse button is down
        else if (Input.GetMouseButton(1))
        {
            //Push the target from our position towards the hit point:
            targetRigidbody.AddForce((targetHitPoint - baseTrans.position).
            normalized * pushForce, ForceMode.Acceleration);
            state = State.Pushing;
        }

        //If neither mouse buttons are held down
        else
            state = State.Idle;
    }

    //If we don't have a target or we have one but it is not in range:
    else
        state = State.Idle;
}
```

The target Rigidbody is accessed so we can call its `AddForce` method. This method takes a `Vector3` for the amount of force to apply, as well as a `ForceMode` enum that defines how the force applies to the Rigidbody.

To apply the force, we use that familiar equation to get the direction we desire:

`(to - from).normalized`

Then we multiply that direction by the force we want to apply, either “pullForce” or “pushForce” based on which one we’re doing.

The `ForceMode` has four values that change two factors of how the force is applied:

- Does it happen as a constant push, like one object pressing against another, or as a sudden impact, like an explosion?
- Is it affected by the mass of the Rigidbody?

Those four possible values are

- **Force**, which is a **constant push** that is **affected by mass**
- **Acceleration**, which is a **constant push** that **ignores mass**
- **Impulse**, which is a **sudden push** that is **affected by mass**
- **VelocityChange**, which is a **sudden push** that **ignores mass**

Our selection, `Acceleration`, ensures that the force we apply is not going to be an instant impulse, as if the object was being hit by a wave of force from an explosion or something of the sort. It is more like a gradual, constant influence pulling it toward us.

It is also ignoring the mass, which means if we pull or push a Rigidbody with a very high mass, the force will still affect the Rigidbody just as much. This makes it so you can make heavy objects and still allow the player to pull and push them.

Aside from applying the force, we also manage the “state” enum so that it always reflects what we were doing during the last `FixedUpdate` call.

With that in place, we can now pull and push Rigidbodies, but we still need to draw our cursor.

Cursor Drawing

We'll make a basic four-pixel (two-by-two) square in the center of the screen that gives the player indication of where their telekinesis ray is being cast from, with a color that responds to the situation.

To do this, all we need is one line of code in an OnGUI event method. As you may remember, we can call GUI methods from the built-in OnGUI event to draw 2D user interface elements to the screen. In our situation, this will be a quick and easy way to draw a simple swatch of color to the screen through code, rather than setting up a Canvas with a UI element for our cursor.

We'll write our OnGUI method **beneath the FixedUpdate method**:

```
void OnGUI()
{
    //Draw a 2x2 rectangle of the CursorColor at the center of the screen:
    UnityEditor.EditorGUI.DrawRect(new Rect(Screen.width * .5f,Screen.
    height * .5f,2,2),CursorColor);
}
```

We're reaching into the UnityEditor namespace to access this method because it's only available through "EditorGUI," not the normal "GUI." You could put a "using UnityEditor;" line at the very top of the script file and cut out the "UnityEditor" part of the reference, if you want, but since we're only using one UnityEditor reference in the script, it won't save us much typing.

One thing to note is that you won't be able to build a game if you're running EditorGUI methods in your game code. The methods are really only meant for use in the Unity editor, which is fine for our purposes. If you were coding for a real game instead of just testing features like we are, you would want to implement the cursor with an actual UI element, like a Panel, and you'd change its color through a reference.

Moving on, the method we're calling is a basic one that just draws a rectangle with a given solid color. It takes a Rect as its first parameter and the Color as the second.

You may remember our usage of the Rect data type (short for rectangle) from our first project:

- The first parameter is the X position of the left side of the rectangle.
- The second parameter is the Y position of the top side of the rectangle.

- The third parameter is the width.
- The fourth parameter is the height.

A value of 0 in the X position is the left edge of the screen, while a value of “Screen.width” would put it all the way at the right edge of the screen.

Similarly, 0 for the Y axis is the bottom edge, while “Screen.height” is the top.

We simply put our rectangle right in the center of the screen by using half of the screen width and height as its position.

The rectangle is not filled in by default – it’s just a rectangular outline, a 1-pixel-thin border. However, if we make it only 2 pixels wide and 2 pixels tall, it will show as a 2×2 square – four pixels in total, all pressed up against each other. It’s small, but we don’t want it to get in the way of what you’re trying to point at anyway, so it will do.

Once this code is in, you’ll be able to see where that ray is coming out of your screen.

To test out the Telekinesis features, try creating three cubes on the ground near the player. Give each one a Rigidbody and give each one a higher mass than the last one. You can make the scale match the mass too, if you want – make the second cube have a scale of (2, 2, 2) and a mass of 2, for example. Then, point at them with the center of your camera and try to pull (left-click) and push (right-click). You’ll see how the Rigidbody takes over the physics, causing the object to turn and bounce as it moves.

Summary

This chapter taught us how to apply external forces to Rigidbodies using the `AddForce` method, as well as the four different options for applying force that Unity provides to us. We also learned that Unity’s physics simulations occur at a fixed timestep, not “once per frame,” and any code that interacts with the physics system constantly should occur in a `FixedUpdate` event, not `Update`.

CHAPTER 38

Moving Platforms

In this chapter, we'll focus on implementing a moving, floating platform that travels back and forth between two locations.

Making a platform move like this isn't very hard. We're experts now. We know how to move Transforms in world space, in local space, and from one point toward another point – we've done all that stuff before.

So the problem we'll face with a moving platform is not getting it to move how we want it to move – rather, it's getting everything sitting on top of the platform to move with it.

It's a somewhat tricky situation. You might expect that having a Rigidbody sitting on top of a moving platform would cause the Rigidbody to move with it. But this isn't always the case.

In Unity, a Transform's motion is either being controlled by the physics system – a Rigidbody – or it's being controlled by us. If it has no Rigidbody attached or it has a kinematic Rigidbody attached, that means we are in control of its movement. It's not “part of the physics simulation” anymore.

Rigidbody's won't react to anything that's not part of the physics simulation. If you have a stack of cubes sitting on top of a platform and those cubes all have a Rigidbody attached, but the platform does not have a Rigidbody, then the cubes aren't going to move when the platform moves. Rather, the platform will slide out from underneath the cubes while they remain still the whole time. Once the platform is no longer underneath them, they'll fall.

But if the object the cubes are sitting on is part of the physics simulation – that is, in control of a Rigidbody – then the cubes will properly react to the movement of that object.

For example, if we threw one Rigidbody-controlled cube at this stack of cubes, causing one of the lower cubes to be shoved by the impact, then the cubes resting atop that one will indeed react to its movement, because there are no kinematic Rigidbody's, there are no scripts moving Transforms of their own accord, and it's just Unity's physics doing its thing.

Scene Setup

To work around this, we'll do a little bit of hacking to get our way. It would be easier to make our platform move by directly setting its Transform position, but as we just established, that isn't going to work for us. We'll have to use a non-kinematic Rigidbody, and we'll have to allow the Rigidbody to move the platform itself.

There are several tweaks we'll have to make to allow a Rigidbody to act as a floating platform without any hiccups or kinks occurring.

Let's create the platform:

- Create a Cube. Name it Moving Platform.
- Set its scale to (16, .3, 16).
- Place it somewhere unobstructed on the X and Z axes and set its Y position to 9 to put it above the ground.
- Add a Rigidbody component.
- I'll make a MovingPlatform material for it, using a dark-blue color with a hex value of 2A2B3D.

This gives us a simple floating platform, like a square plate hovering above the ground, shown in Figure [38-1](#).

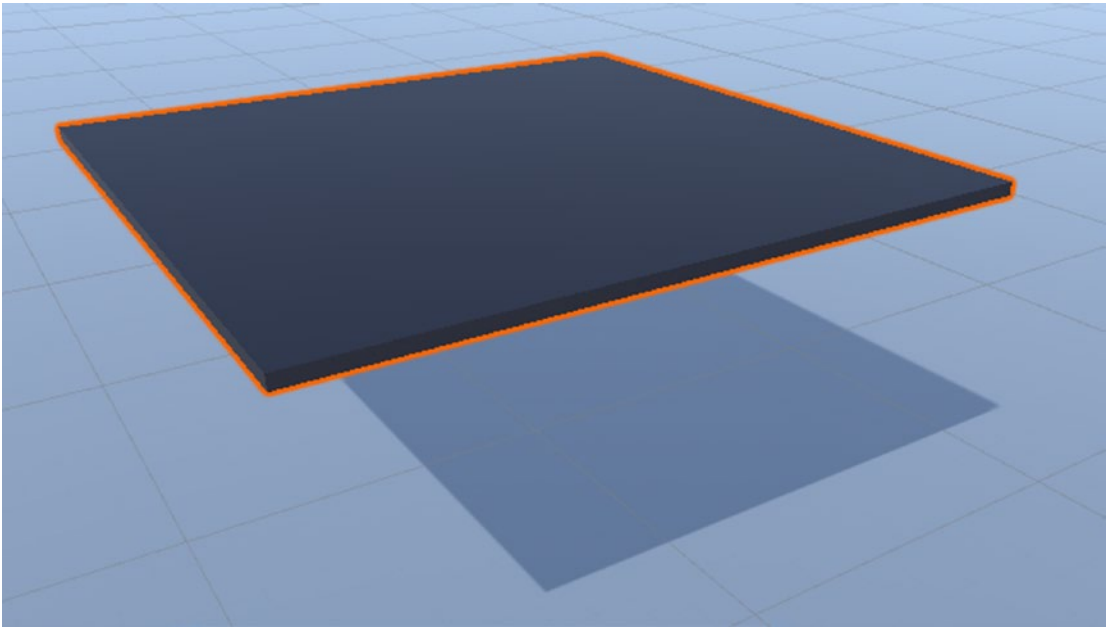


Figure 38-1. *The Moving Platform hovering over the floor*

Now we need to use the Inspector to tweak some fields for our Rigidbody component. First, we know we want the platform to float, so we'll uncheck the "Use Gravity" field. This means it won't gather downward momentum when midair.

It will, however, still be pushed downward when Rigidbodies resting on top of it are driving their weight against it – and they will be, since they'll be subject to gravity. To counter this, we could constrain the Rigidbody's position so that its Y axis is frozen – but this means we won't be able to make a platform that moves on the Y axis. Instead, we'll simply give the platform a very high mass to ensure that the objects on top of it won't budge it. In the Mass field, set the value to $1\text{e}+09$. You can also type 1000000000 – that's nine zeroes – to get the same value. This is the maximum mass a Rigidbody can have.

We also want to make sure that the platform does not tilt or twist as a result of the weight on top of it. We'll do that with the Constraints field, which contains two fields inside it: Freeze Position and Freeze Rotation. Each axis has a checkbox that can be ticked to ensure that the Rigidbody does not change the object's position or rotation for that axis. Check all three boxes for the Freeze Rotation field, ensuring our platform does not rotate (unless we script it to).

When you're done, your Rigidbody component should look like Figure 38-2.

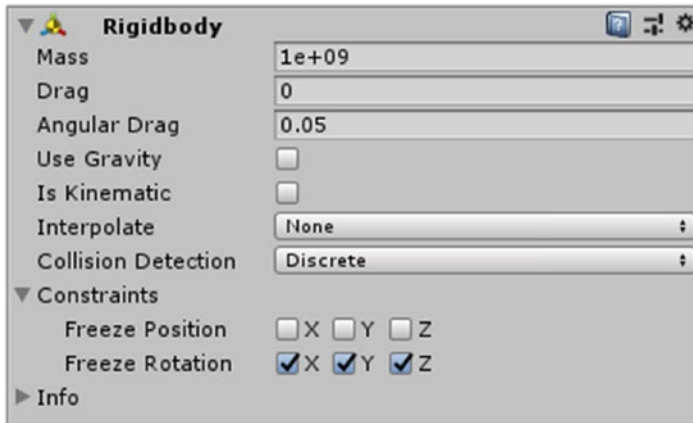


Figure 38-2. The Rigidbody component of our Moving Platform in the Inspector. It has maximum mass, does not use gravity, and has rotation frozen for all three axes

Platform Movement

Let's get the platform movement working, and then we'll test it by placing some cubes on it. We'll interact with the Rigidbody.velocity member to move the platform rather than directly moving the Transform.

The Rigidbody.velocity is a Vector3 depicting the velocity that the Rigidbody is moving per second. This is not modified by the mass, meaning it's the actual amount of movement that the Rigidbody expects to be occurring per second.

Normally, you wouldn't modify the "velocity" directly. You would apply forces with AddForce and let Unity handle the velocity itself. But we need a finer degree of control over our platform movement, so we're going to break that rule.

Our platform will be positioned where we want it to start at. It will travel back and forth from this initial position to a Vector3 location that we'll set in the Inspector. Every time it reaches its destination – either going to the target position or going back to the initial position – it will wait a given amount of time before starting on the way back.

Let's start by creating a PlatformMovement script and declaring our variables:

```
public class PlatformMovement : MonoBehaviour
{
    private enum State
    {
        Stationary,
        MovingToTarget,
        MovingToInitial
    }

    [Header("References")]
    [Tooltip("The Transform of the platform.")]
    public Transform trans;

    [Tooltip("The Rigidbody of the platform.")]
    public Rigidbody rb;

    [Header("Stats")]
    [Tooltip("World-space position the platform should move to.")]
    public Vector3 targetPosition;

    [Tooltip("Amount of time taken to move from one position to the other.")]
    public float timeToChangePosition = 3;

    [Tooltip("Time to wait after moving to a new position, before beginning to move to the next position.")]
    public float stationaryTime = 1f;

    //Returns the units to travel per second when moving.
    private float TravelSpeed
    {
        get
        {
            //Distance between the two positions, divided by number of seconds taken to change position:
            return Vector3.Distance(initialPosition,targetPosition) / timeToChangePosition;
        }
    }
}
```

```

    //Gets the current position we're moving towards based on state.
    private Vector3 CurrentDestination
    {
        get
        {
            if (state == State.MovingToInitial)
                return initialPosition;
            else
                return targetPosition;
        }
    }

    //Gets the current distance from our position to the current
    destination.
    private float DistanceToDestination
    {
        get
        {
            return Vector3.Distance(trans.position, CurrentDestination);
        }
    }

    //World position of platform on Start.
    private Vector3 initialPosition;

    //Current state of the platform.
    private State state = State.Stationary;

    //State for the platform to use next - either MovingToTarget or
    MovingToInitial.
    private State nextState = State.MovingToTarget;
}

```

The only variable that might not have a clear purpose to you is probably the “nextState”. This variable is used to store the state value we want to switch to after we wait the “stationaryTime”. Since our “state” will be set to Stationary during this time, we need a second State variable to store whether we must move toward the target position or the initial position once stationary time ends.

Let's move on to declare our `Start` and `FixedUpdate` methods, as well as a method that we can invoke to switch to the next state. We'll put it all **down below the variables we just declared**:

```
//Transitions 'state' to the 'nextState'.
void GoToNextState()
{
    state = nextState;
}

//Unity events:
void Start()
{
    //Mark the position of the platform at start:
    initialPosition = trans.position;

    //Invoke the first transition in state after 'stationaryTime' seconds:
    Invoke("GoToNextState",stationaryTime);
}

void FixedUpdate()
{
    if (state != State.Stationary)
    {
        //Set velocity to travel from our position towards the current destination
        by 'TravelSpeed' per second:
        rb.velocity = (CurrentDestination - trans.position).normalized *
        TravelSpeed;

        //Calculate how much distance our velocity is going to move us this frame:
        float distanceMovedThisFrame = (rb.velocity * Time.deltaTime).magnitude;

        //If the distance we'll move this Update is enough to reach the
        destination:
        if (distanceMovedThisFrame >= DistanceToDestination)
        {
            //Reset velocity to zero and snap us to the position so we don't
            overshoot it:
```

```

        rb.velocity = Vector3.zero;
        trans.position = CurrentDestination;

        //Based on our current state, determine what the next state
        will be:
        if (state == State.MovingToInitial)
            nextState = State.MovingToTarget;
        else
            nextState = State.MovingToInitial;

        //Become stationary and invoke the transition to the next state
        in 'stationaryTime' seconds:
        state = State.Stationary;
        Invoke("GoToNextState",stationaryTime);
    }
}
else //If we are stationary
    //Maintain velocity at 0 to prevent unwanted movement:
    rb.velocity = Vector3.zero;
}

```

The Start method sets our “initialPosition” right away so we know where the platform is supposed to return to after reaching its “targetPosition”. It also Invokes our “GoToNextState” method, which is how we switch our “state” from Stationary to whatever we last set our “nextState” value to. This Invoke in the Start method kicks off the repeating process. The default “state” is Stationary, so something must trigger our first movement toward the target position. That’s why the default value of “nextState” is set to MovingToTarget when we declare the variable.

Our FixedUpdate handles the movement towards the current destination while our “state” is not Stationary.

The value we apply to “velocity” is familiar – we’ve been over moving an object toward another many times before, so I’m sure you recognize the “(to – from). normalized” equation.

Our method of determining if we’re about to hit our destination is a simple equation getting the movement our velocity is going to take us on this frame (velocity multiplied by deltaTime) and then calculating the magnitude of that vector. That gets us the distance we’re going to travel on this frame. Since we know our velocity is pointing us at

the current destination, all we need to do is compare “the distance to our destination” with “the distance we’re traveling on this frame.” If we’re traveling an equal or greater distance, we’ll be reaching or overshooting our target this frame. In that case, we can initiate the switch to Stationary, snap our position to the destination (to avoid any overshooting), and Invoke the switch to the next state. Before we switch our “state” value, though, we use it to determine what our “nextState” should be. If “state” is `MovingToTarget`, then our next state needs to take us back to the initial position, and vice versa.

With that, let’s get it up and running. Save your code and attach a `PlatformMovement` script to our Moving Platform GameObject. Set the “trans” and “rb” to the Transform and Rigidbody components of the same GameObject. For the “targetPosition”, we can write in the same value as our current position and then add a little bit to one of the axes. I’ll add 15 points to the Z axis.

Figure 38-3 shows the `PlatformMovement` script in the Inspector, assuming its initial position is at the world origin, but with a Y value of 9.

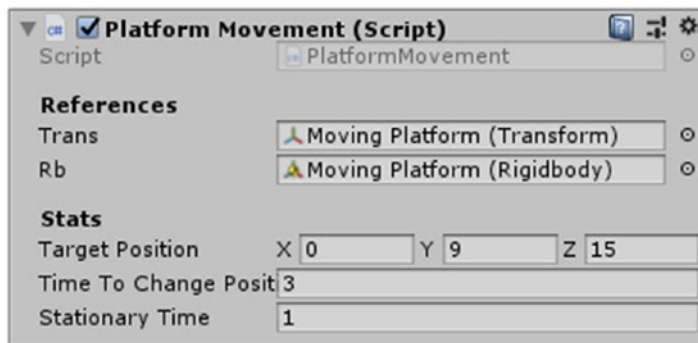


Figure 38-3. *The PlatformMovement script in the Inspector*

If you don’t want to find and view the platform from the player’s perspective, you can deactivate the root Player GameObject for now so that it doesn’t take mouse focus when the game starts and simply view the platform in your Scene window instead.

Either way, if you observe the platform while the game is playing, you’ll see it meander back and forth between its points as we expect it to. But all that fussing with using a Rigidbody was to ensure that other Rigidbodies move when they’re on top of the platform, so let’s add two cubes stacked atop each other and test that:

- Create a cube named Big Cube. Scale it to (6, 6, 6) and drag it up until it's a little above the platform. Just make sure it's not sticking into the platform, which could cause some unsavory results. The Rigidbody gravity will drag it down and make it touch anyway.
- Add a Rigidbody and give it a Mass of 6.
- Copy-paste the Big Cube and rename the copy to Little Cube. Set its scale to (2, 2, 2) and the Rigidbody mass to 2. Drag it so it's above the Big Cube, but again, not quite touching it.

When you're done, your setup should look something like Figure 38-4.

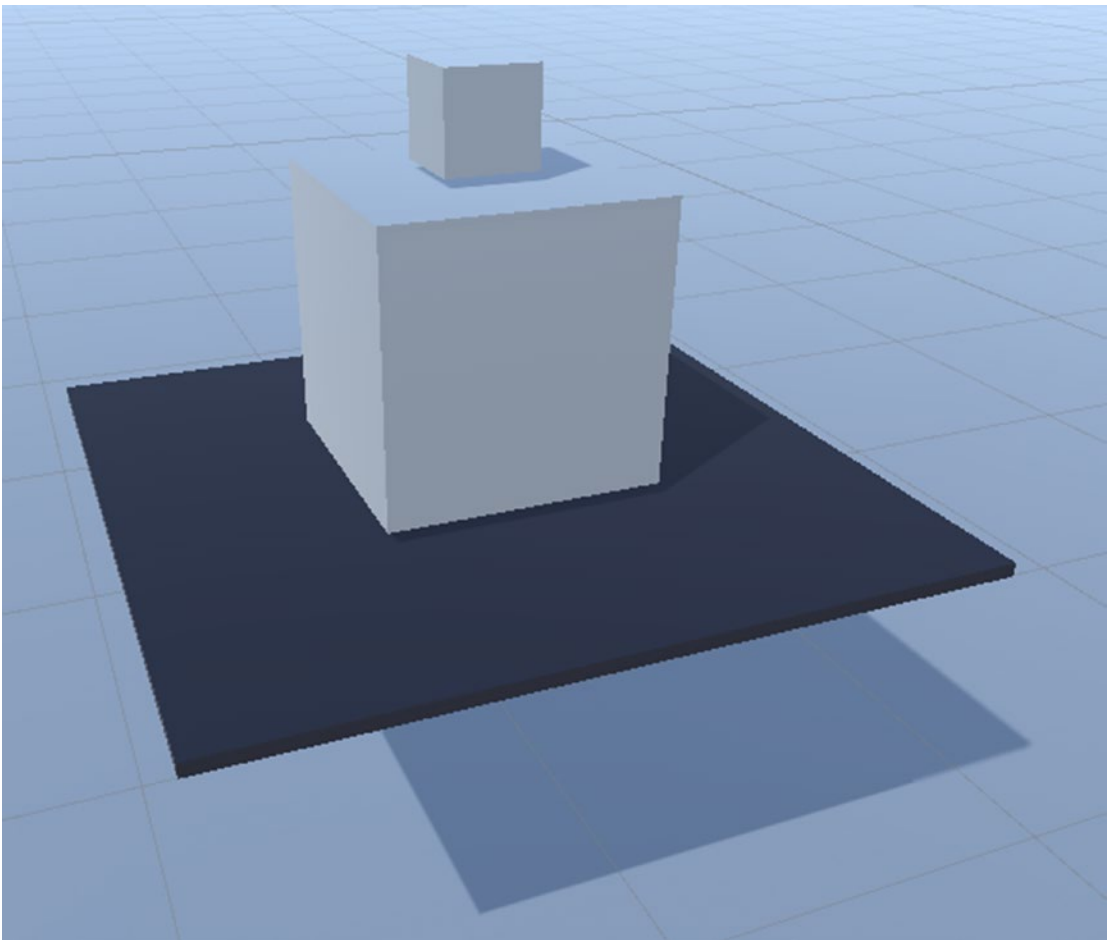


Figure 38-4. *Our Moving Platform with the Big Cube and Little Cube resting on top of it*

If you test it out, you'll see that the cubes do indeed move with the platform, but they'll likely not last very long before toppling over. The sudden change in velocity when the platform starts and stops gives them a jolt. If this were a problem for us, we would likely want to implement some smoothing to the starting and stopping of movement.

The platform will also stop if it hits any static colliders, like a wall with a collider but no Rigidbody. Since it waits to reach its destination before turning back, it can get stuck this way. Generally, you'll want to keep the platform in a path that won't be obstructed by anything. Since it has such high mass, it should be able to shove other Rigidbodies out of its way, but any static Collider (no Rigidbody attached) will block it indefinitely.

Player Platforming

Now that the platform is moving and other Rigidbodies on top of it are moving with it, we just need to make our player attach to the platform when they land on it.

We'll be creating two scripts to handle this:

- A **Platform** script will be attached to any GameObject that the player should "stick to" when standing on.
- A **PlatformDetector** script will be attached to any GameObject that is not controlled by a Rigidbody, but that should stick to platforms it lands on. This will be attached to the Player.

Create both of those scripts right now, one named Platform and one named PlatformDetector. We want both of their classes to exist before we begin.

First off, open the Platform script and fill it with this code:

```
public class Platform : MonoBehaviour
{
    void OnTriggerEnter(Collider other)
    {
        //Try to find a PlatformDetector on the touching object:
        PlatformDetector detector = other.GetComponent<PlatformDetector>();

        //If there is a detector,
        if (detector != null)
            //Set this Transform as its 'platform' variable:
            detector.platform = transform;
    }
}
```

```

void OnTriggerExit(Collider other)
{
    //Try to find a PlatformDetector on the touching object:
    PlatformDetector detector = other.GetComponent<PlatformDetector>();

    //If there is a detector,
    if (detector != null)
        //Null out its 'platform' variable:
        detector.platform = null;
}
}

```

This simply sets the PlatformDetector’s “platform” variable to the Transform of the platform whenever a GameObject that has a detector touches the Platform. Conversely, whenever the PlatformDetector stops touching the Platform, its “platform” variable is set to null.

Now open your PlatformDetector and give it this code:

```

public class PlatformDetector : MonoBehaviour
{
    [Tooltip("Transform to move with the platform.")]
    public Transform trans;

    //The Transform of the platform we are currently standing on, if any:
    [HideInInspector] public Transform platform = null;

    //Position of the platform on the last Update.
    private Vector3 platformPreviousPosition;

    //True if we have set the position of the platform at least once since
    it was first set.
    //False if we have not yet set the position of the current platform.
    private bool firstPositionLogged = false;

    //Unity events:
    void FixedUpdate()
    {
        //If we are standing on a platform
        if (platform != null)

```

```

{
    //If we have already logged the platform position at least once
    and it is not the same as its current position
    if (firstPositionLogged && platformPreviousPosition !=
        platform.position)
    {
        //Add the change in platform position to our trans.position:
        trans.position += platform.position - platformPreviousPosition;
    }

    //Log the platform position this frame:
    platformPreviousPosition = platform.position;
    firstPositionLogged = true;
    //Mark that we have logged the position at least one
}
else //If we are not standing on a platform
{
    //We'll mark that we have not set the platform's position yet.
    //When a new platform is assigned, we won't move the transform
    until this is set to 'true'.
    firstPositionLogged = false;
}
}
}

```

This is a simple little system that tracks the position of the Platform at the end of the last FixedUpdate, as long as there is a platform. Each Update, it compares the platform's current position to the platform's last position. Whatever the difference is, it adds that to the position of the Transform referenced in the Inspector for the script, which can be our Player.

This effectively makes our attached Transform move by whatever the platform has moved each FixedUpdate.

But in order to ensure that the first FixedUpdate that occurs after we acquire a platform does not compare the platform position to the default value of “platformPreviousPosition”, we must use this “firstPositionLogged” bool. If we didn't do this, then the very first frame that the “platform” variable was set during would use

a “platformPreviousPosition” that has not yet been updated. It could be at the default value of `Vector3.zero`, or it could be at whatever it was set to last when we had a different platform under us. This would create a very incorrect difference in the two positions during the first frame that we land on a platform, which could cause some very strange movements to the Transform using the `PlatformDetector`.

With that, we need to make the Player contain a trigger collider that will trip the Platform and, of course, add a `PlatformDetector` as well. We’ll cover the bottom of our player with a little trigger collider that will touch platforms we stand on:

- Add a Rigidbody to the root Player Transform and set it to kinematic.
- Add a Sphere Collider and check the Is Trigger box. Set its Radius to 1 to match our `CharacterController` and raise it up with a Center Y value of .8 so it pokes down off the bottom of the player just a bit.

Now, be sure to add a Platform script to your existing Moving Platform GameObject. With that, you should be able to hop on the platform with the Player and watch them move along with it. Once you step or jump off, you’ll become detached and no longer share its movement.

Summary

In this chapter, we highlighted the difference between GameObjects that are controlled by the Unity physics system – those with non-kinematic Rigidbodies attached – and those that are controlled by scripts editing their Transform position. We implemented our platform movement using a Rigidbody rather than a Transform to ensure that other Rigidbodies react to its movement. We also defined a means of automatically attaching the player, which is not controlled by a Rigidbody, to the platform.

CHAPTER 39

Joints and Swings

This chapter will go over an example of some basic usage of a physics component called the Configurable Joint, which can be used to bind GameObjects together with the physics system. We'll use it to create a chain of objects, each one attached to the one above it to create something like a rope. At the bottom of the rope, we'll attach a platform that will swing with the rope. Our player can use their "telekinesis" to pull the swing or its joints and then jump on the platform to ride it.

Swing Setup

Let's set up the GameObjects of our swing first so they're all ready to be connected by joints. The swing will consist of a hovering cube that remains stationary and cannot be pulled or pushed. Beneath this cube, three identical "chain links" will hang below. Each chain link is a sphere with a long, slender cube hanging beneath it. Each sphere will connect to the cube of the chain link above it. The finished product is shown in Figure 39-1.

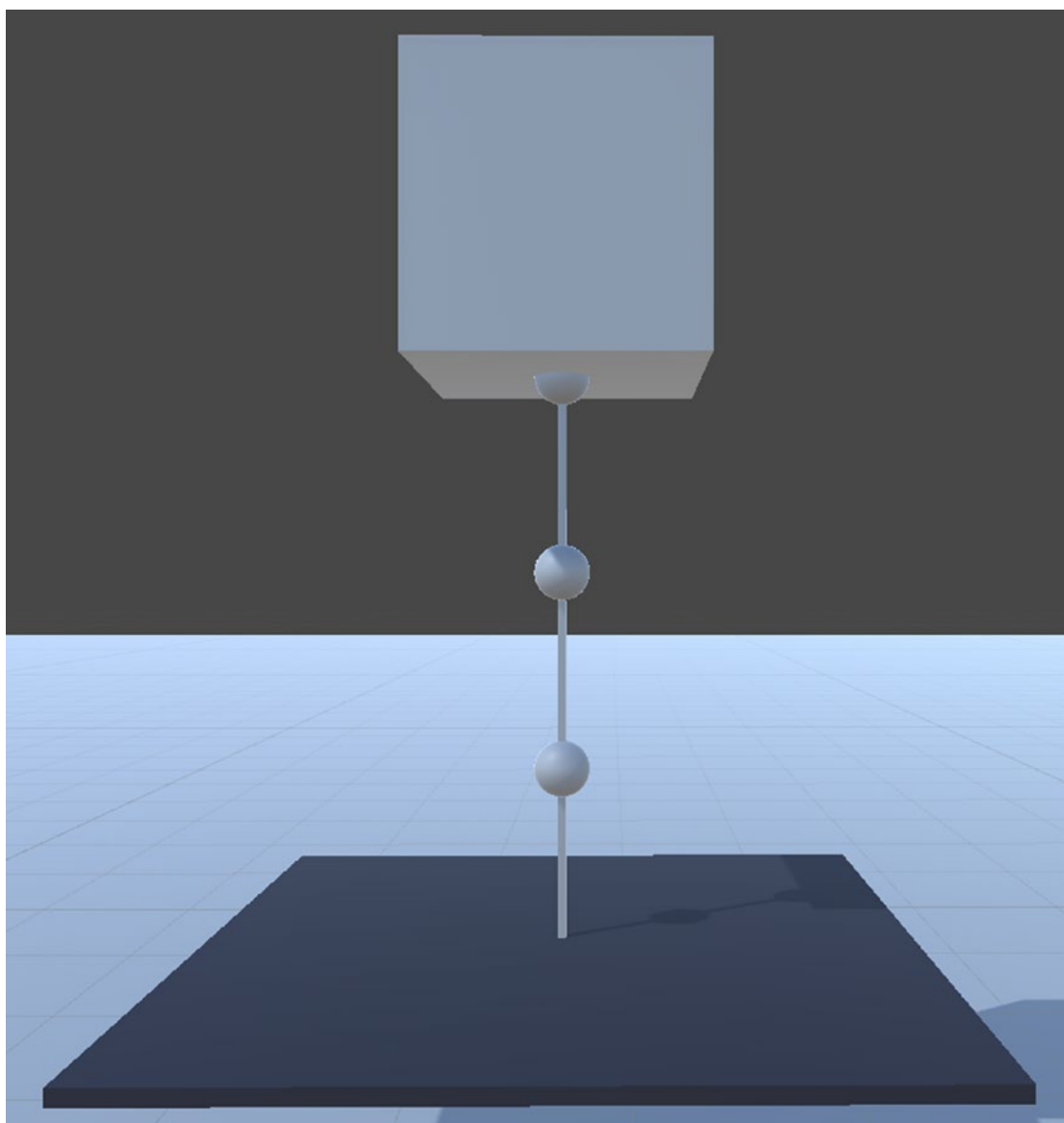


Figure 39-1. The full Swing GameObject hovering above the floor, consisting of the Hovering Cube on top, three Chain Links hanging below it, and a Platform cube connected at the bottom

Start by creating an empty GameObject named “Swing” with no parent. We’ll use this as a root for the Swing. We’ll be able to select it to move the whole Swing and all its pieces wherever we want.

You can set up the Swing wherever you want on the X and Z axes – find a clear place in your Scene and set it there. Keep its Y position set to 0.

- Add a Cube child to the Swing. Name it Hovering Cube. Set its scale to (10, 10, 10) and its local position to (0, 35, 0). Set its layer to “10: Unmovable”.
- Add an empty child GameObject to the Swing. Name it Chain Link and set its local position to (0, 30, 0).
- Add a Sphere child to the Chain Link. Leave its local position at (0, 0, 0) and set its scale to (2, 2, 2).
- Add a Cube child, also to the Chain Link. Set its scale to (.3, 5, .3) to make it slender and long. Set its local position to (0, -3.5, 0) to place it just beneath our Sphere.

After we set this one Chain Link up, we’ll be copy-pasting it to create the others. But first, let’s make sure it has the components we need so we aren’t adding them and setting them up three times.

We’ll need each Chain Link root GameObject (not the Sphere or Cube within) to have a Rigidbody and a Configurable Joint added. Go ahead and add each of those now and give the Rigidbody a Mass value of 1.5.

You might think we should add a Rigidbody not to our Chain Link, but to both the Sphere and the Cube within it, but this isn’t necessary.

The Rigidbody of the Chain Link will detect the Box Collider and Sphere Collider of its child GameObjects and will consider them fused together to form a single body, acting as if both of those colliders are part of the same object. We can effectively think of them as two pieces of metal welded together. If the Cube is struck by something, then it moves and the Sphere pivots with it – and vice versa.

These are known as **compound colliders** and can be used to represent a more complex object by shaping it out of “primitive” collider types. Primitive colliders are the colliders for basic built-in shapes: Box Collider, Sphere Collider, and Capsule Collider. By creating a parent GameObject with a Rigidbody attached, then adding children with primitive colliders, we create one whole object with a more complex shape than just a cube, sphere, or capsule.

You can do this to “summarize” the shape of a more complex mesh with a combination of primitive shapes.

Moving on to our Configurable Joint, we have a few values to set.

The Configurable Joint component is attached to the GameObject that you want to connect another GameObject to, and the Rigidbody of the other GameObject is referenced in the Configurable Joint. So when making a chain out of these Chain Link objects, this means that the Configurable Joint will be attached to the upper (higher) link in the chain, and the link beneath that chain will be referenced as the “Connected Body” member (the first member listed in the Inspector).

The Anchor value of the Configurable Joint is a Vector3 depicting from where the Connected Body pivots. The location is local to the Transform of the GameObject with the Configurable Joint component. We want our Anchor to be at a value of (0, -6, 0).

The position of the anchor is shown in the Scene as a dark-gray set of position handles. With our Anchor setting, this will place it at the bottom of the cube, shown in Figure 39-2.

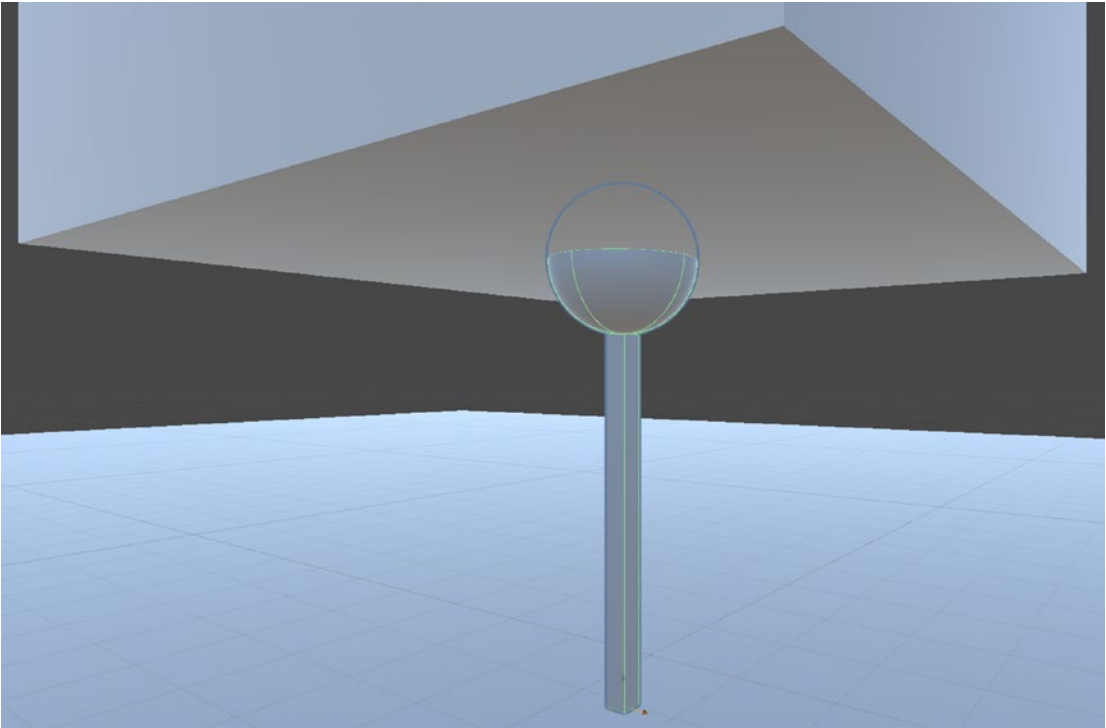


Figure 39-2. The Chain Link is shown sticking out of the bottom of our Hovering Cube. The Configurable Joint Anchor location is visible at the bottom of the Chain Link Cube as a small set of arrows

This will ensure that the Chain Link we position below this one will be pivoting around the bottom of this link, not the center of the Sphere (which would be quite awkward).

Moving on, the six “Motion” and “Angular Motion” dropdown fields in the Configurable Joint are all that’s left to set.

Each of these fields represents a single axis: X, Y, or Z. Each one can be set to Locked, Limited, or Free. The Motion fields represent whether the Rigidbody can change position on that axis. The Angular Motion fields represent whether the Rigidbody can rotate on that axis:

- When **Locked**, the axis is not changed by the joint at all.
- When **Limited**, the axis is affected, but limited by the other fields that can be customized down below.
- When **Free**, the axis can move as much as warranted with no limitations.

For our purposes, we’ll set all three of the Motion fields to Locked because we don’t want the joints to cause movement, just rotation – they pivot around each other.

As for the Angular Motion fields

- If we allow **angular X motion**, the chain links can swing **forward and back**.
- If we allow **angular Y motion**, the chain links can **twist sideways**, allowing the platform to turn.
- If we allow angular Z motion, the chain links can swing **right and left**.

The swing will be more controlled if you only allow it to pivot on the X or Z axis and lock the other two. The Y axis isn’t all too important, simply depicting whether we’ll see the cubes twisting and turning from the forces applied to them.

For our tests, let’s set the X angular motion to Free and set both the Y and Z angular motions to Locked. This will allow it to swing forward and back only, something like a pendulum. Since this is all local to the Transform, if you want it to swing left or right instead, you can just rotate the entire Swing on the Y axis instead of changing the Configurable Joint settings.

The rest of the fields beneath the Motion fields won’t need any tampering. At this point, Figure 39-3 shows how your Rigidbody and Configurable Joint components should look in the Inspector.

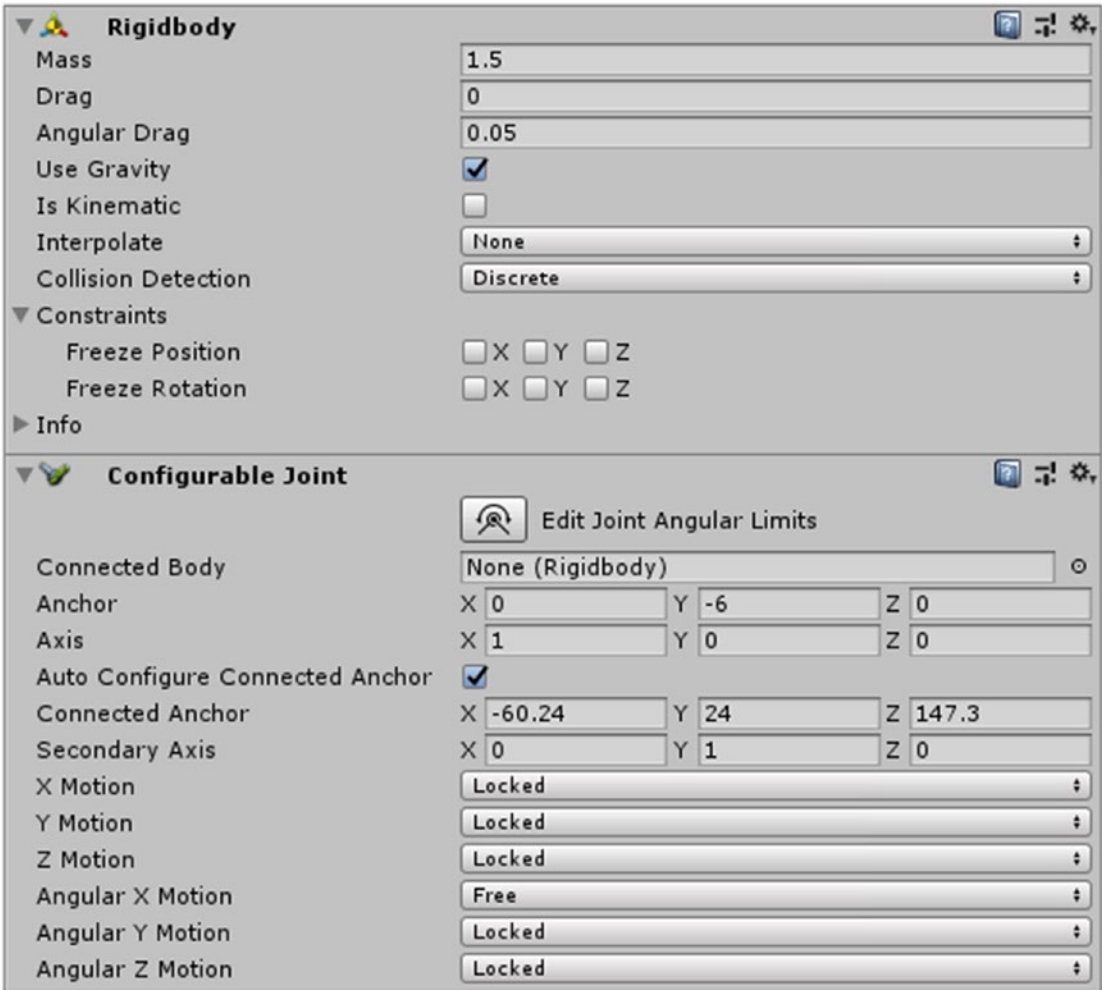


Figure 39-3. Our Rigidbody and Configurable Joint components shown in the Inspector

Now, let’s continue the setup:

- Select the Chain Link and copy-paste it. With the copy selected, decrease its Y position by 7 units in the Inspector. You can do this by simply typing a “- 7” after the current Y position value in the field, and Unity will calculate the new value as soon as you click away from the field. This will position the second Chain Link so that the bottom of its Sphere is just beneath the Cube of the upper link. If you’d rather type the value in yourself, it should be a Y position of 23.

- Again, copy and paste the Chain Link we just made, and apply the same -7 units to the Y axis so the Y position is 16.

We now have three Chain Links, all touching at their tips to create – you guessed it – a chain. They stick out the bottom of our Hovering Cube, as shown in Figure 39-4.

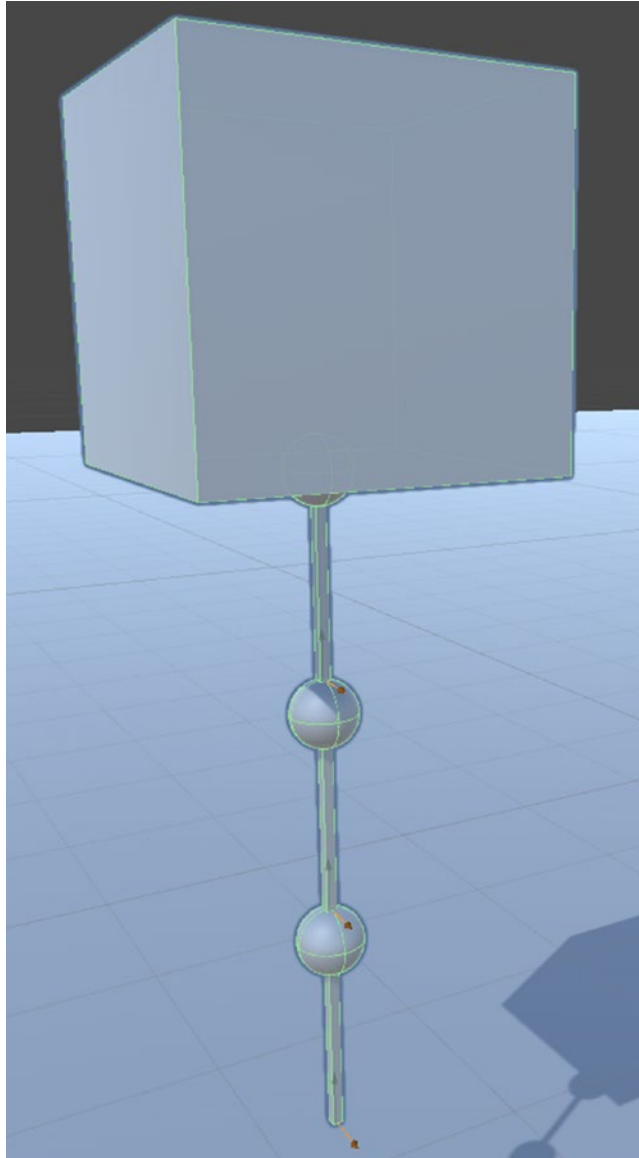


Figure 39-4. Our entire Swing object while selected. Each Configurable Joint Anchor is also shown at the bottom of each Cube

We just need a platform at the bottom of the “chain” now:

- Add a Cube child to the Swing GameObject. Name it Platform. Set the scale to (25, .5, 25) and position it at (0, 9.75, 0) so it’s just beneath the lowest Chain Link.
- Add a Rigidbody component to the Platform. To prevent the platform from colliding with the player and tilting about, check all three Freeze Rotation boxes. Set Mass to 4 and Drag to .15.
- I’ll add the existing MovingPlatform material to the cube as well, giving it a dark-blue color.

With that, our setup of the bits and pieces of our swing is complete, which means your swing should at last look like it does in Figure 39-1 from before.

Connecting the Joints

In order to connect the joints to each other, we’ll first want to connect the upmost Chain Link to the Hovering Cube itself.

To add a Configurable Joint component, you will have to add a Rigidbody as well. Trying to add one without a Rigidbody already attached will result in Unity automatically adding one.

Since we don’t want our Hovering Cube to be affected by any collisions or forces, we’ll constrain its position and rotation through the Rigidbody. Check all six boxes under the Constraints field of the Rigidbody in the Inspector. You can also uncheck the Use Gravity field, since it wouldn’t make much sense to apply gravity when we’ve already frozen our Rigidbody in place.

The Rigidbody settings should look like Figure 39-5 when you’re done.

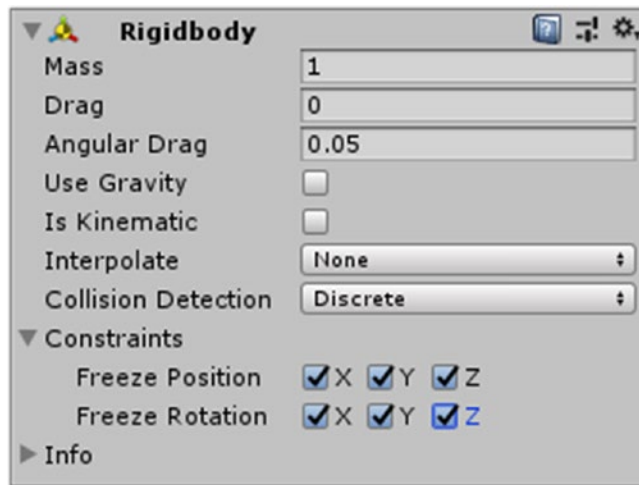


Figure 39-5. *Rigidbody settings in the Inspector for our Hovering Cube*

Now we have the Hovering Cube anchored in place above the ground, so if we attach our first Chain Link to it by a Configurable Joint (which we'll do in a second), the link will be bound to the Hovering Cube by that joint, keeping it from falling freely through the air. With the other links attached to that one, they'll all hang off the Hovering Cube.

With a Configurable Joint added to your Hovering Cube, don't forget to set the Anchor to the bottom of the cube with a value of (0, -5, 0). Remember, the Anchor is local to the Transform, so each unit in the Anchor is multiplied by the Transform scale. For our Chain Links, the scale was (1, 1, 1) so the Anchor value was effectively in world units, but since our Hovering Cube is scaled to (10, 10, 10), that means each unit in the Anchor setting is worth 10 world space units. That's why it's -5 to put it at the bottom of the Cube, not -0.5. Think of it as "50% of the height of the cube," not ".5 units".

We also need to lock our Motion values in the Configurable Joint of the Hovering Cube. Following the same settings you used on the Chain Links, shown before in Figure 39-3, apply them again to the Hovering Cube: set all six fields to Locked except for the Angular X Motion, which should remain set to Free.

Now let's attach each link to the one above it. Remember, the "Connected Body" field of the Configurable Joint should be set to the link that is lower to the ground. If you haven't changed the Chain Link names after copy-pasting them, their names should be as follows:

- **Chain Link** is the highest one.
- **Chain Link (1)** is the middle one.
- **Chain Link (2)** is the lowest one.

So first, select the Hovering Cube and drag Chain Link (the topmost one) from the Hierarchy onto the “Connected Body” field of the Configurable Joint. This binds the first Chain Link to the Hovering Cube.

Now we can go down the chain and attach each lower link to the upper link:

- Chain Link should have its Connected Body set to the Rigidbody of Chain Link (1).
- Chain Link (1) should be set to Chain Link (2).
- Finally, Chain Link (2) should have the Platform Rigidbody as its Connected Body, which binds the Platform to the bottom link.

Finishing Touches

You should now be able to play the game, run over to your Swing with the Player, and try pulling and pushing the Platform or the Chain Links with the Telekinesis feature. Remember, the swing has been set to only swing on the X axis, which is forward and backward. This means it’s like a pendulum, swaying back and forth in only one direction. Trying to pull and push it from the wrong side won’t generate much of a reaction.

With our current gravity settings, the swinging may seem a bit “off.” Unity’s default gravity settings are tweaked to appear realistic when the unit of measurement for your game is 1 meter. As you’ll recall from our earliest chapters, what a single unit resembles is totally relative. Since our player is 6 units tall, we’re really using a unit measurement of about 1 foot. The gravity setting thus thinks we’re a bit larger than we really are: the player would be 6 meters tall by that standard, not 6 feet! This can account for the swing moving in a somewhat “slow-motion” way.

As with many things, you can change the force that gravity applies in the Edit ► Project Settings window. The gravity is the first field shown when the Physics tab is selected, as depicted in Figure 39-6.

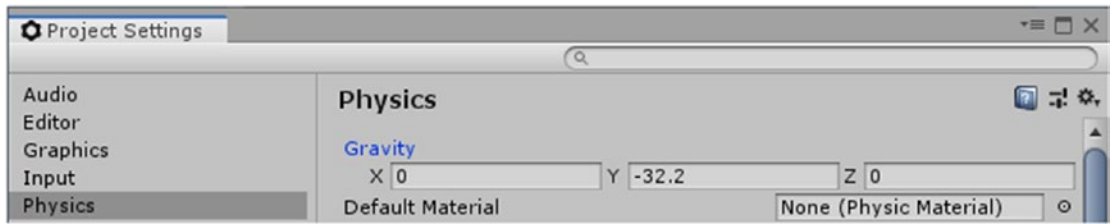


Figure 39-6. In the Project Settings window, the Physics tab is selected, exposing the Gravity field

This shows a Vector3 for the gravity force applied to Rigidbodies constantly (per second). Of course, the X and Z axes are set to 0 since gravity does not standardly pull you in such directions. The Y axis is what we want.

The default setting of -9.81 mimics the pull of gravity on Earth but using meters as a measurement. Since a meter is about 3.28 feet, we can multiply this default setting of 9.81 by 3.28, which is roughly 32.2. Of course, it should be -32.2 since we want it to be downward force.

Once you've set the gravity this way, your swing movement should look more natural.

One final touch is to add a Platform script to our Platform cube GameObject. Once you've done that, try setting the swing in motion with Telekinesis and then hopping onto it. The Player should stick to the swing and move with it.

As expected, since everything involved in the motion of the Swing is driven by the physics system, other Rigidbodies will stay on the swing and move with it as well. Try stacking some Rigidbody cubes on top of the swing and then set it in motion with Telekinesis.

Summary

This chapter taught us how to use the Configurable Joint component to attach two Rigidbodies such that they pivot around each other. Some key points to remember are as follows:

- Box Colliders, Sphere Colliders, and Capsule Colliders are considered **primitive** collider types. These are the most basic and cheap collider types.

- A parent `GameObject` with a `Rigidbody` attached will consider all its children with primitive `Collider` components to be part of the same whole object. When one collider is struck, it's as if they were all struck, since the `Rigidbody` considers them one attached unit.
- The `Configurable Joint` component should be attached to the `GameObject` to which the other `Rigidbody` is attached. The "Connected Body" field refers to the `Rigidbody` that should be attached to the `GameObject` with the `Configurable Joint` component.

CHAPTER 40

Force Fields and Jump Pads

In this chapter, we'll be implementing a configurable ForceField script that pairs with a trigger collider to apply force to Rigidbodies and/or the player. Based on a variable set in the Inspector determining the type of force to apply, we'll either apply it constantly while the trigger is touched or once when the trigger is first entered. Based on the force mode you choose, you could make an object either get pushed as if by a large fan or get shoved up like a "jump pad" thrusting them all at once.

Script Setup

Let's start by creating a script named ForceField in the Scripts folder and declaring our variables:

```
public class ForceField : MonoBehaviour
{
    [Tooltip("Should the force field affect the player?")]
    public bool affectsPlayer = true;

    [Tooltip("Should the force field affect Rigidbodies?")]
    public bool affectsRigidbodies = true;

    [Tooltip("Method of applying force.")]
    public ForceMode forceMode;

    [Tooltip("Amount of force applied.")]
    public Vector3 force;
```

```

[Tooltip("Should the force be applied in world space or local space
relative to this Transform's facing?")]
public Space forceSpace = Space.World;

//Gets the force in world space.
public Vector3 ForceInWorldSpace
{
    get
    {
        //If it's world-space we can just return 'force' as-is:
        if (forceSpace == Space.World)
            return force;

        //If it's local space, we use our transform to convert 'force'
        from local to world space:
        else
            return transform.TransformDirection(force);
    }
}

```

This gives us a basic setup allowing each script to control how it behaves with some tweaking of the variables in the Inspector. We can make the force field affect only the player, only Rigidbodies, or both. We can change the force mode, the amount of force applied, and whether the force is applied locally (`Space.Self`) or in world space (`Space.World`). We also have a property giving us a quick way to grab the amount of force to apply, automatically converting it to a world space direction relative to the Transform that the ForceField is attached to if the “forceSpace” is `Space.Self`. This is done with the `Transform.TransformDirection` method, which takes a `Vector3` and converts it from being local to the Transform to instead be world space.

This way, instead of using “force” directly, we can use the “ForceInWorldSpace” when calling `Rigidbody.AddForce`, and we’ll know it’s applying the force in the correct direction, in world space (which is what `Rigidbody.AddForce` expects).

When we learned how to pull and push Rigidbodies for our Telekinesis script, we went over the four different ForceMode settings. To sum it up, the **Force** and **Acceleration** modes apply constant force, where **Force** is affected by the Rigidbody mass and **Acceleration** is not. Conversely, **Impulse** and **VelocityChange** modes apply a sudden shove, where **Impulse** is affected by Rigidbody mass and **VelocityChange** is not.

Force Field Setup

Let's set up a simple ForceField GameObject so it's ready to roll when we've finished coding it:

- Start by creating an empty GameObject as a root. Name it Force Field. Set its layer to ForceField. Add a Rigidbody, mark it as kinematic, and add the ForceField script.
- Add a Cube child to the Force Field. Leave its scale at (1, 1, 1) and set its local position to (0, .5, 0). This puts the bottom of the cube at the position of the root Force Field GameObject.
- Check the “Is Trigger” box for the Cube's Box Collider component.
- Create a material named Force Field in your Materials folder. Change the first field, Rendering Mode, to Transparent. I'll apply a green-blue color with a hex value of AAFEE3 and give it transparency by setting the alpha to 25. Apply the material to the Cube we made.
- Drag and drop the Force Field root GameObject from the Hierarchy to the Prefabs folder in the Project view to create a prefab.

This gives us a simple little semi-transparent box with a trigger collider attached, shown in Figure 40-1. We can use the prefab to create a force field of any size. Scaling up the root Transform will increase the size of the force field while keeping it at the same position on the ground.

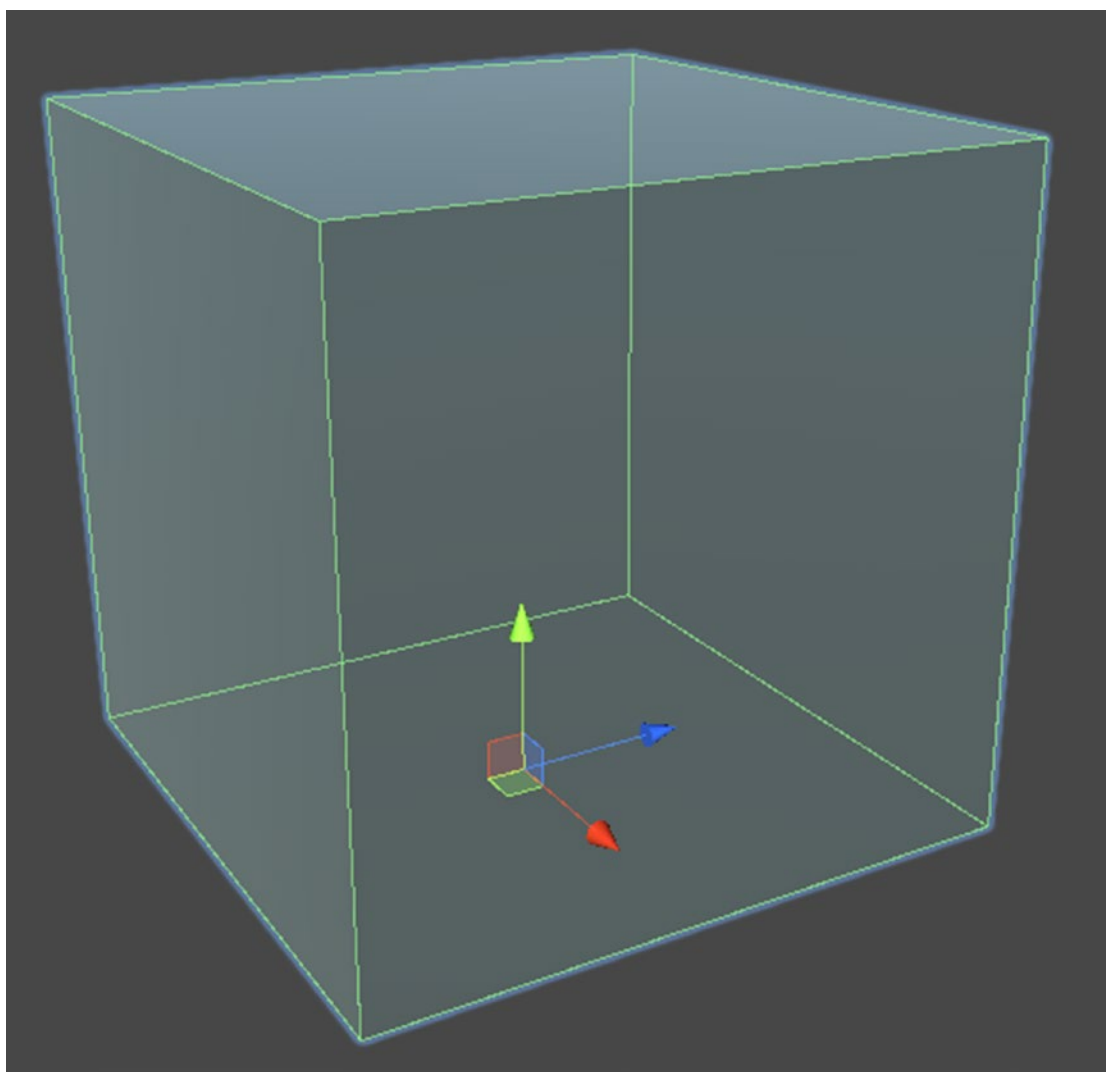


Figure 40-1. The Force Field GameObject

Adding Velocity to the Player

Before we make the ForceField apply its force, we still need some method of applying force to our Player. Since the Player doesn't use a Rigidbody, we must implement this ourselves. Luckily, it won't be so hard at all. We'll add this method **in our Player script**, **just beneath our old WallIsNearby method**:

```

public void AddVelocity(Vector3 amount)
{
    //Add the velocity X and Z to our 'worldVelocity':
    worldVelocity += new Vector3(amount.x,0,amount.z);

    //Add the velocity Y to our 'yVelocity':
    yVelocity += amount.y;

    //Ensure that we become midair if our Y velocity was raised above 0.
    //If we don't do this, it will be set to -1 again in ApplyVelocity if
    we are grounded.
    if (yVelocity > 0)
        grounded = false;
}

```

This method exposes a simple process of adding velocity in all three axes to the Player, working with our existing system. You'll recall that the Player "worldVelocity" is a Vector3, but it is for the X and Z axes only, keeping a value of 0 at its Y axis. The "yVelocity" float handles the Y axis instead. Thus, we can't just add the velocity to our "worldVelocity", so we create a new Vector3 that applies only the X and Z axes when we add the velocity to "worldVelocity". We then separately add the "amount.y" to our "yVelocity".

You might recall that, in the ApplyVelocity method which makes the Player move by its velocity per second, we constantly apply a -1 "yVelocity" while the Player is grounded. Thus, if we had a ForceField shoving the player upward when the player is already grounded, it would immediately be overridden by the Player setting their velocity back to -1 before moving. That's why we must make sure we set "grounded" to false if the Y velocity has become a positive value.

Applying Forces

To apply forces to touching objects, we'll need two separate trigger collider events. OnTriggerStay will handle the Force and Acceleration force modes, making them apply their force constantly at a rate of "force" per second while the trigger is first touched. OnTriggerEnter will handle the Impulse and VelocityChange force modes, making them apply "force" as is, not per second, but only once when the trigger is first touched.

Each of these methods provides us with a single parameter pointing at the other Collider that was touched.

Let's declare those methods and their contents **within the ForceField script class**:

```
void OnCollisionTouched(Collider other)
{
    //If we affect the player,
    if (affectsPlayer)
    {
        // check for a Player component on the other collider's GameObject:
        var player = other.GetComponent<Player>();

        //If we found one, call AddVelocity:
        if (player != null)
        {
            //If the force mode is a constant push mode, use Time.deltaTime
            to make the force "per second".
            if (forceMode == ForceMode.Force || forceMode == ForceMode.
            Acceleration)
                player.AddVelocity(ForceInWorldSpace * Time.deltaTime);
            else //Otherwise, use the force as-is.
                player.AddVelocity(ForceInWorldSpace);
        }
    }

    //If we affect Rigidbodies,
    if (affectsRigidbodies)
    {
        // check for a Rigidbody component on the other collider's GameObject:
        var rb = other.GetComponent<Rigidbody>();

        //If we found one, call AddForce:
        if (rb != null)
            rb.AddForce(ForceInWorldSpace, forceMode);
    }
}
```

```

void OnTriggerEnter(Collider other)
{
    //Impulse and VelocityChange modes will apply force only when the
    trigger is first entered.
    if (forceMode == ForceMode.Impulse || forceMode == ForceMode.VelocityChange)
        OnColliderTouched(other);
}

void OnTriggerStay(Collider other)
{
    //Acceleration and Force modes will apply force constantly as long as
    the collision stays in contact.
    if (forceMode == ForceMode.Acceleration || forceMode == ForceMode.Force)
        OnColliderTouched(other);
}

```

We detect the collider being touched with an Enter and a Stay method, and each one calls the same “OnColliderTouched” method we declared in the preceding code.

This method checks for a Player component on the GameObject of the touching Collider if the “affectsPlayer” bool is true, and if it finds one, it adds force to the player. Whether or not we use Time.deltaTime in the force depends on the ForceMode, since Force and Acceleration modes are expected to apply “per second” through the OnTriggerStay method, while Impulse and VelocityChange are expected to apply only once through the OnTriggerEnter method.

Roughly the same thing is done with Rigidbodies: if the “affectsRigidbodies” bool is true and there is a Rigidbody attached, apply force. For Rigidbodies, we don’t need to use Time.deltaTime to apply the constant force. The Rigidbody.AddForce method will automatically look at the force as “per second” if we use either the Force or Acceleration mode.

With that, you can set up some force fields in your Scene and try them out. They’re somewhat small by default, but we can set their scale value to whatever we want or simply use the scale tool (hotkey R) to scale them by eye.

Set the “force” vector to something noticeable: try a Y value of 100, for example. Using Force or Acceleration mode will cause the touching entities to hover upward while they remain in the force field. Using Impulse or VelocityChange will apply the force all at once, causing the entity to jerk upward instead. This would be useful for making a “jump pad” of sorts, allowing the player to reach high places by stepping on it or to launch objects far distances by pushing or pulling them onto it with Telekinesis.

You can also place force fields sideways on walls and switch the Force Mode to Self to apply the force relative to the Force Field facing direction. Just remember to keep track of where your axes are pointing when you set the “force” variable. The arrows shown by the position tool (hotkey W) when you select your Force Field will point in the local direction of each axis, so you can know which axis to apply force in. If the arrows are pointing along the world direction, just press the X hotkey to make them local to the selected Transform. For example, Figure 40-2 shows a Force Field on the side of a wall, with the position tool arrows showing local to the Force Field. In this case, if you want the Force Field to shove entities away from the wall, you would set your force as the Y axis (the green arrow) of the “force” vector.

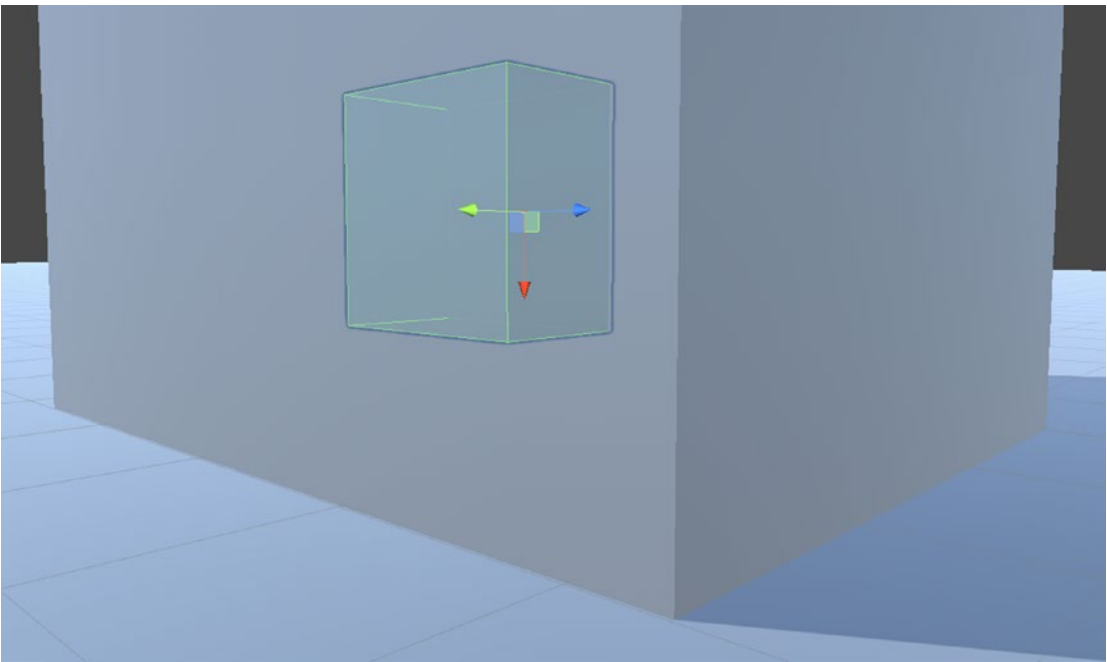


Figure 40-2. A Force Field is shown sticking off the side of a wall, with its Y axis pointing away from the wall

Summary

In this chapter, we implemented a Force Field script that can be used for a sudden shove, like a jump pad for our player or a constant force. We combined our knowledge of adding forces to Rigidbodies, as well as trigger colliders and their associated collision detection events: `OnTriggerStay` and `OnTriggerEnter`.

CHAPTER 41

Conclusion

This chapter marks the conclusion of our final example project and this book. We’ve come a long way and learned a lot since we started, but don’t think of getting bored yet. C# is an old language rich with features, and we’ve hardly demonstrated all that the Unity game engine is capable of. There’s plenty more to learn. Don’t stop seeking out new information and adding tools to your belt!

Let’s get a summary of what we went over with this project, and then we’ll part with some ideas on where to take your learning from here. Even if you decide not to structure your learning very rigidly past this point, just trying to implement new things can take you a long way. If you’ve followed this book from start to finish, you ought to have an understanding of the environment you’re working in that allows you some room to wiggle and try out stuff you’re really interested in. You might not have as much guidance this way, but working out solutions yourself can help you become better at problem solving. Keep trying new things, and if you get in over your head, take a step back and reevaluate. You can always come back to a lofty project some time down the road, when you’ve learned new things and become a better programmer – and working on things you care about most will help to keep your interest piqued.

Physics Playground Recap

This project gives us experience dealing with Unity’s 3D physics system, as well as some new concepts on how to deal with vectors and 3D motion. Here’s a summary of the major takeaways:

- The **magnitude** of a vector is a float value depicting how much distance the vector travels. It can also be called the **length** of a vector. The “magnitude” member of Vector3 is a property that returns the magnitude.

- **Normalizing** a vector makes it point in the same direction it did before, but only have a magnitude of 1. This means that multiplying a normalized vector by a float X will travel X units in the direction the vector points.
- Physics updates are not synchronized with normal frame updates. Each physics update will occur at a set interval, with the default being **50 updates per second** (once every .02 seconds). If necessary, they might happen multiple times per frame, or they might happen only once every three frames – it all depends on the framerate. The gist of it is that the updates will occur 50 times per second whether the game is running slow (low framerate, choppy) or fast (high framerate, smooth).
- Scripts which interact on every frame with physics components like the Rigidbody should use the **FixedUpdate** built-in event method, not Update. FixedUpdate occurs every physics update, so 50 times per second by default.
- If a GameObject is to be controlled by a Rigidbody, you should not move its Transform directly through scripts. Instead, add forces with the Rigidbody.AddForce method.
- The Rigidbody.velocity property can be used to get or set the velocity of a Rigidbody through script. It's not recommended to set the velocity every FixedUpdate unless you have a special use case for it. You generally want to be using the AddForce method.
- For Rigidbodies to be carried along by the movement of another GameObject, such as a moving platform, the movement of all objects involved must be part of the physics system. Moving a Transform directly will not cause Rigidbody-controlled GameObjects resting on top of it to move with it.
- **Primitive** collider types are the colliders for basic shapes: Box Collider, Sphere Collider, and Capsule Collider.

- To create a Rigidbody with a shape defined by multiple primitive colliders, use a root GameObject with a Rigidbody component attached, and then attach Colliders to child GameObjects. This is a **compound collider**. Only the root GameObject needs a Rigidbody component. All Colliders will count as part of the same whole object controlled by the root Rigidbody.

Further Learning for Unity

Let's go over some loose ends pertaining to the Unity engine that you may be interested in checking out. These are things we didn't get into during our projects to keep things from getting bloated. If you're interested in learning about a feature of the engine, the Unity Manual pages on the official Unity site offer a good starting point.

Your favorite search engine should have an easy time guiding you to Manual pages. Just search for "unity manual" and tack on the feature you're interested in at the end, for example, "unity manual terrains."

The Asset Store

The Unity Asset Store can be accessed using the Asset Store window in the Unity editor or by visiting it on the Web (a quick search should find it). The Asset Store is a place where other Unity developers can upload products for use within the engine, including scripts for code extensions and game functionality, 3D models or 2D graphics, music, sound effects, and so on. As the name depicts, anything that could be used as an asset in your Project window could be placed on the Asset Store. You can download these assets into your project right from the window in the Unity editor.

The Asset Store contains both free and paid assets. If you're in need of something that you can't quite make yourself, you might just be able to find it there.

Terrains

Unity has a built-in system for creating and editing 3D terrains. You add the terrain to the Scene as a flat surface and then use tools to change the shape, allowing you to form hills and valleys. You can then add textures, which are images that get drawn onto the terrain.

These images might represent grass, rock, dirt, and so on. You can mix textures, blending the images together to create the look of a gradual transition between different types of ground. There's also support for adding trees and grass that sway in the wind.

Coroutines

The coroutine is a concept that can be employed to provide some timing-related functionality for method calls that you can't quite achieve with Invoke calls. A coroutine is a method that can be "yielded" with a special line of code. Yielding can stop execution of the code for some given amount of time – a number of seconds or until the next frame or until some condition is true. After the wait, the code continues execution at the same line, with all the same local variables in the same state. This can be useful for handling more delicate processes without the need for invoking many different methods. It can also be used to perform a strenuous task bit by bit to prevent from generating a drop in framerate or a lengthy pause in gameplay: a loop with very many iterations can "yield" until next frame every five to ten iterations so that it "runs in the background" instead of doing it all at once.

The concept of coroutines in programming is not necessarily specific to Unity, but Unity has its own implementation of coroutines unique to the engine. The `MonoBehaviour` class (base class for all script components) is the entry point for calling coroutines, after all.

Script Execution Order

Under the Edit ► Project Settings window, there is a tab called Script Execution Order which we never had a reason to use in our example projects. It may prove useful to you at some point in the future, so it's good to know what it's for.

The name entails its purpose pretty well: it allows you to specify a consistent order in which your scripts receive event calls like Update or Start. You can make the event calls for one script always occur before another if the order is important in some special way.

Further Learning for C#

Polishing up your skills with the programming language you use can be a good way to expand your horizons. Knowing all you're capable of doing with the language may guide you to solutions you wouldn't have found otherwise. This section will provide an overview of some of the features we didn't get to go over in detail with our example projects.

Delegates

Delegates provide a means of declaring a variable that can point at methods, something like a way to reference a method as if it were an object instance. The variable can be called like a method, without knowing exactly which method is attached to it.

To sum up the process

- Declare a delegate, giving it a name, return type, and any number of parameters. This is like a template or a blueprint that methods must follow.
- You can now declare a variable that uses the delegate name as the variable type.
- Any method which matches the delegate return type and parameters can be assigned to the variable. The variable can then be called like a method.

Let's review the syntax involved in each of these steps:

```
//Declare the delegate like a method but with the 'delegate' keyword before.
//We provide a return type (string), name (MyDelegate), and two parameters.
delegate string MyDelegate(string a, int b);

//You can now declare a variable whose type is the delegate name.
MyDelegate delegateVariable;

//Now declare a method that matches the return type and parameters of the
delegate.
string AddNumberToString(string a, int b)
{
    return a + b;
}
```

```
//Since the method matches the delegate return type and parameters, we can
assign the method to the variable.
delegateVariable = AddNumberToString;

//The delegate variable can be called like a method.
delegateVariable("Hello World",1);
```

In this example scenario, you might as well just call the method by its name directly, of course. An actual use case might be a custom UI system where a Button class exposes a delegate variable that is called when the button is pressed. When creating a button, we can supply any method that matches the delegate to be called when the button is pressed, making it easy to reuse the button for different purposes. We could even change the method that the delegate variable points to on the fly to change what occurs when the button is pressed.

Documentation Comments

C# defines a system of “documentation comments” that can be written before definitions such as classes, methods, variables, properties, and so on. These comments use a special syntax to define documentation for code, right within the lines of the code. Code editors and other software can read the comments and use their contents in useful ways. One purpose for this that you’ll see used widely is the “summary” tag, which uses documentation comments to place a description of a definition which will then pop up when you mouse over that definition anywhere in the code editor.

For example, you might write a summary before a class you declared. Sometime later, when you declare a variable with the class name as its type, you can mouse over the class name and see that description you wrote.

In fact, Unity provides a summary description for most of their built-in classes. You’ve probably seen them when you leave your mouse hovering over a type or method name for a second in Visual Studio Code.

The following code shows how to write a basic “summary” for one of our existing methods: the `WallIsNearby` method we declared for our `Player` in the last example project. Documentation comments always start with three slashes `///` instead of the two `/*` that make up a normal comment. They contain “tags” inside them: an opening

tag, like “<summary>”, and then a closing tag of the same name but with a slash before it, like “</summary>”. The text that belongs to the tag goes between those two tags:

```
/// <summary>
/// Checks if a wall is near enough to the player for them to wall jump off
/// of it.
/// Returns true if there is a wall, false if there is not.
/// </summary>
private bool WallIsNearby()
{
    ...
}
```

As you can see, it always starts with “///”, with a single space coming after the three slashes (because it looks prettier that way). We start the “<summary>” tag, write the text we wish to serve as the summary, and then write the closing tag “</summary>”.

Documentation comments can also provide other kinds of data that is used in other ways. Methods can have a description provided for each of their parameters so that while you write out a method call, the description of each parameter is shown to you as you type. This is done with the “<param>” tag. Methods can also have a “<returns>” tag that gives a description of what the method returns (assuming it doesn’t return void).

Here is an example of “param” and “returns” tags, using our (somewhat useless) method we declared for our delegate a little bit ago:

```
/// <summary>
/// Describe what the method does here.
/// </summary>
/// <param name="a">Describe parameter 'a' here.</param>
/// <param name="b">Describe parameter 'b' here.</param>
/// <returns>Describe what the method returns here.</returns>
string AddNumberToString(string a, int b)
{
    return a + b;
}
```

This shows how to declare a parameter within a tag: the param tag has a “name=” passage within that is used to specify the name of the associated parameter.

You may be interested to know that this method of writing and formatting data with this “tag” syntax is widely used to represent various different kinds of data through text files, and is known as XML: Extensible Markup Language. It’s a flexible and readable way to declare text data.

In favor of keeping our code samples a bit less bulky, we haven’t employed this custom throughout this book. However, you’ll often see it if you look at other people’s C# code. It can be useful to maintain since it allows you to get a description of a method on the fly just by hovering your mouse over it. This can make it easier to work with code you wrote a while ago, where the details on how to use it might have since slipped your mind. For example, you might forget what a method returns or the purpose of each parameter it declares. Having that information pop up as you type can make things clearer. It can also be useful when writing code that others are meant to use. We may not be able to go check what a built-in Unity method does or what a class is for, so it’s nice to have a summary when we mouse over it.

Writing these documentation comments out for every declaration sounds tedious, though, doesn’t it? Luckily, most code editing software will automatically create relevant documentation comments when you type “///” before a method declaration. But since Visual Studio Code relies mostly on extensions to provide language-specific features, you won’t find this functionality out of the box.

At the time of writing, there is an extension called “C# XML Documentation Comments” which can be installed to automatically add documentation comments like this. Other extensions likely exist which do the same thing too. If you forgot how to find and install extensions in Visual Studio Code, take a quick peek back at Chapter 1.

Exceptions

Many programming languages have control structures in place that revolve around reacting when code that you’re running throws an exception. “Exception” is the C# word for “error.” Any code that results in an error popping up in your Console in Unity is “throwing an exception.”

Exceptions are actual data types that all inherit from a base class `Exception`, which is in the `System` namespace. When an exception is thrown, it’s done with a simple line of code: the “throw” keyword and then a constructor that creates an `Exception` instance. The most basic form is to use the base `Exception` class:


```

void DoSomething()
{
    if (a)
        throw new Exception("Failed to do something.");
    else
    {
        //...
    }
}

```

However, more particular types exist that inherit from `Exception`, which make it clearer what went wrong and why the error occurred. For example, `IndexOutOfRangeException` occurs when trying to get an index from a collection (array, `List`, etc.) that does not contain an item at that index.

A control structure that you'll commonly see when coding is the "try...catch...finally":

```

try
{
    //Run some code you think might produce an exception here in the 'try'
    block.
}
catch (Exception e)
{
    //If an exception occurs in the 'try', this code will run and the
    Exception will be 'caught' so it isn't thrown as a message in the
    Console.
    //The 'Exception e' parameter will point to the Exception that was
    thrown, which may contain useful data.
}
finally
{
    //This code runs after the fact, no matter whether an exception
    occurred or not.
}

```

This can be used to define “fallback code” for what should occur if an exception was thrown while some code was running. Whatever type the “catch” block declares as its parameter, that’s the type of Exception that will be handled. You can even declare multiple “catch” blocks, each one catching a different type of Exception. If your “try” block results in some Exception being thrown that is of a type that none of your “catch” blocks are expecting, the Exception will show up in the Console and is considered “uncaught” or “unhandled.” But as long as the Exception is caught by one of your “catch” blocks, it won’t end up in the Console.

Advanced C#

If you feel confident and want to further explore the nitty-gritty details of C#, this section will list some extra features and talking points that you may be interested in learning about, briefly explaining the basic concept so you can explore it on your own after.

Operator Overloading

Operator overloads are something like methods that can be declared in a class to write your own code that occurs when a certain kind of operator is used with your class instance. This allows you to, for example, allow two instances of your class to be operated on by a “+” operator. You could also allow some other type to be added to your class with the “+” operator, like an int or string. When the operator occurs, it’ll be your code running and generating the result based on the operands (which is the technical term for “the values on either side of the operator”).

Conversions

As well as overloading operators like “+”, “-”, “*”, and so on, there are also “conversion operators” that you can declare within a class. They allow you to write code that determines what is returned when your class type is implicitly or explicitly converted to another type. If you have some data structures that are similar to each other, you can declare conversion operators to allow easy conversion from one type to the other.

An example of such a conversion within Unity’s built-in types is that of Vector2 and Vector3. The Vector2 type is just a vector with only an X and Y value. The Vector3 is pretty much the same, but it has the Z axis as well. A Vector2 can implicitly convert to a

Vector3 – implicit meaning we don’t have to tell it to convert, it just does it if necessary. A Vector3 can explicitly convert to a Vector2 – explicit meaning we must tell it to convert by using a conversion operator, because data will be lost when we convert it.

Both cases were implemented with a conversion operator declared in each vector class, specifying what to return when performing the conversion.

Generic Types

Generics are a somewhat confusing but very powerful feature. We already lightly explored generics during this book, but to give true justice to this concept, you may want to do some further digging. Learning how to declare classes and methods that use generic types can be a good way to gain further understanding so you don’t just know how to work with generic types, but how to use them in your own definitions.

Structs

We went over the difference between classes and structs before, but we never learned how to declare structs and when you might consider using them.

Generally speaking, structs will give you more trouble than classes. If you don’t quite know how to use them, you’ll have your compiler throwing errors at you in no time. They can be a bit of a pain. That being said, knowing when and how to use them can get you a few steps ahead of the game.

Summary

Our final chapter has given you some ideas for further learning to pursue on your own, as well as some basic demonstrations. Here’s a short summary:

- A **coroutine** is a method call that can **yield** on the spot, stopping its operation for a given amount of time and then resuming in the same state. This means you can yield in the middle of a loop and then continue at that same point in the loop.
- A **delegate** is something like a way to store a reference to a method in a variable.

- An **exception** is the C# word for “error.” When code **throws** an exception, that means something went wrong and generated an error, like those seen in the Console window of Unity.
- Data types like classes and structs can declare **overloads** to allow specified **operators** (like + and –) to be used with the type or to allow **conversions** to other types.

With these ideas in mind, it’s now up to you to decide where to go – isn’t that exciting? The concepts discussed briefly in this chapter are a good place to start. The more tools you have on your belt and the more experience you have with using them, the easier it will be to find clever solutions to the problems you face. If you encounter new syntax, don’t be afraid to look for information on it and figure out how it works!

Index

Symbols

&& operator, [86](#)
> and < operators, [85](#)
== operator, [83](#), [127](#)
|| operator, [85](#)

A

Abstract method, [335](#), [337](#), [440](#)
Access modifier, [91](#)
AddForce method, [518](#)
Advanced C#
 conversion, [562](#), [563](#)
 generics, [563](#)
 operator overloading, [562](#)
 structs, [563](#)
Agent Height, [402](#)
Agent Radius, [402](#)
AimAtTarget method, [354](#)
Alignment field, [370](#)
Anchor elements, [367](#)
ApplyVelocity method, [549](#)
Arcing projectile
 AnimationCurve, [426](#), [427](#)
 colliders, [431](#)
 curve editor, [428](#)
 Explode method, [430](#)
 FractionOfDistanceTraveled
 member, [426](#)
 Lerp call, [429](#)

 OnSetup method, [427](#)
 script creation, [425](#)
Armor / damage, [443](#)
ArmorType, [116](#)
Arrays
 currentPointIndex variable, [211](#)
 indexing, [210](#)
 parentheses () method, [210](#)
 square brace set [], [211](#)
Arrow GameObject, [339](#)
Arrow key movement, [318](#), [319](#)
Arrow Tower Button, [370](#)
Arrow towers, [351–354](#), [404](#)
Assets folder, [12](#), [20](#), [58](#), [131](#), [147](#), [313](#), [452](#)
Assignment operator, [73](#)
Asynchronous operation, [276](#)
AsyncOperation method, [271](#), [273](#), [277](#)
Attributes, [110–111](#)
Awake() event, [180](#)
Axes, [23](#), [48](#)

B

Barricades, [437](#), [438](#)
bool alive variable, [330](#)
Box colliders, [340](#), [342](#), [343](#)
BoxCollider/SphereCollider, [440](#)
Breakpoints
 buttons, [134](#)
 control panel, [134](#)
 DebuggingTest, [132](#)

INDEX

Breakpoints (*cont.*)

- GameObject, 132

- variable box, 133

buildButtonPanel, 410

Build buttons, 369

Build mode

- buttons, 362

- first towers, 362

- functionality, 394

- position, 362

- UI, 361

- user interface, 361

Build mode-logic, 379

- camera, 382

- gold indicator, 380

- highlighter, 380, 381

- screen position, 380

BuildModeLogic() method, 380, 391, 411

Buisld Mode settings, 379

Build settings

- menu, 300

- scene, 206

- Unity editor, 207

- Unity Hub program, 301

- window, 205

Build Support module, 300

BuildTower method, 387, 388

Built-in method, 375

Button component, 390

C

C#

- delegates, 557, 558

- document comments, 558–560

- exceptions, 560, 561

CalculatePath method, 406

Calling methods, 67, 68

Camera component, 375

Camera movement, 315

- apply, 320–322

- methods, 317, 318

- mouse drag, 322, 323

- setting up, 315, 316

- start/update, 316

- target position, 323

- zooming, 324, 325

CancelInvoke method, 414

CanDashNow property, 255

Cannonballs, 425

Cannon tower prefab

- ArcingProjectile script, 431

- build button, 433

- FiringTower script, 434, 435

- OnClick event, 433

Canvas GameObject, 363

Canvas Scaler component, 365

CharacterController

- applying references, 167

- collisions, 167

- component, 166, 167

- method, 168

- Quarternion, 169

- slerp, 169

Checkpoints, 306

Child GameObjects

- dragged cube, 30

- hierarchy window, 29

- highlighted object, 31

- indentation, 31

- movement/rotation/scaling, 31

- parent-child relationship, 31

- transform, 29

Classes, 67, 87

- accessing members, 90–93

- constructor, 98

- declaring, [87](#)
 - declaring constructors, [95–98](#)
 - instance methods, [93–95](#)
 - variables, [89, 90](#)
 - Class statement, [62](#)
 - Code blocks, [61, 63](#)
 - Code editors
 - blank space, [6](#)
 - breakpoints, [7](#)
 - color themes, [5](#)
 - debugger, [7](#)
 - downloading, [5](#)
 - extension, [6](#)
 - Microsoft Visual Studio, [4](#)
 - Collider[] array, [499](#)
 - Collider type, [340](#)
 - Collision detection
 - colliders, [181](#)
 - collision matrix, layer, [187](#)
 - kinematic movement, [182](#)
 - layers, [183, 184, 186](#)
 - Rigidbody, [182, 197](#)
 - Color field, [152](#)
 - Comments, [63, 64](#)
 - Compiling, [57](#)
 - Complex pathing, [444](#)
 - Component
 - camera, [16](#)
 - functionality, [16](#)
 - GameObject, [14, 15](#)
 - sliders, [16](#)
 - Compound colliders, [535](#)
 - Conditions
 - AND operators, [86](#)
 - else block, [81, 82](#)
 - else if block, [82, 83](#)
 - enum, [80](#)
 - equality operators, [83, 84](#)
 - greater than/less than
 - operators, [85](#)
 - if block, [77, 78](#)
 - OR operators, [85](#)
 - overloads, [79](#)
 - Configurable Joint component, [543, 544](#)
 - Console window, [58, 78](#)
 - const keyword, [231](#)
 - Constructor chaining
 - armor, [120](#)
 - base, [119](#)
 - currentDurability, [119, 121](#)
 - dealsBluntDamage member, [121](#)
 - equipment class, [119](#)
 - item class code, [118](#)
 - maxDurability, [119](#)
 - parameters, [118](#)
 - Conversion operators, [562](#)
 - Coroutine, [556](#)
 - Creating plane, [24](#)
 - Cube Base, [35](#)
 - Cube Middle, [35](#)
 - Cube Top, [35](#)
 - CurrentCornerIsFinal
 - property, [418](#)
 - Current Gold Panel, [371](#)
 - currentGoldText, [378](#)
 - cursorIsOverStage, [378](#)
 - CursorLockMode.Locked, [464](#)
- ## D
- Dashing
 - cooldown, [255–257](#)
 - methods, [251–253](#)
 - public void Die() method, [254](#)
 - variables, [249–251](#)

INDEX

- Data types, [69](#)
- Death method
 - CharacterController, [178](#)
 - invoke, [177](#)
 - SetActive method, [178](#)
- Debug.DrawLine method, [67](#)
- Debugging
 - breakpoint (*see* Breakpoints)
 - environment prompt box, [131](#)
 - launch.json file, [130](#)
 - run button, [130](#)
 - unity documentation, [135](#), [136](#)
 - Unity editor, [131](#)
 - Update method, [129](#)
- Debug.Log method, [69](#), [137](#)
- Delegates, [557](#), [563](#)
- DeselectBuildButton()
 - method, [381](#), [391](#)
- DeselectTower() method, [381](#), [389](#), [390](#)
- Designing levels
 - adding walls, [263](#)
 - camera GameObject, [264](#)
 - creation, [261–263](#)
 - directional light, [262](#), [264](#)
 - player prefab instance, [262](#)
 - prefabs/variant, [259–261](#)
- Detecting, patrol points
 - class types, [217](#)
 - GetComponent<T> method, [216](#)
 - for loop, [218–221](#), [230](#)
 - list, [217](#)
 - List<Transform>, [218](#)
 - script, [216](#)
- Dictionary, [385](#)
 - Containskey, [386](#)
 - private variables, [385](#)
 - towers, [388](#)
- DontDestroyOnLoad method, [276](#)

E

- Editor extension
 - accessing inspectors, [237](#)
 - CustomEditor attribute, [237](#)
 - drawing scene, [238–240](#)
 - scripts, [236](#)
- Else statement, [62](#)
- Enabling / disabling
 - folded player script
 - component, [175](#)
 - inactive GameObject, [175](#)
 - initialization code, [174](#)
 - Player GameObject,
 - checkbox, [174](#)
 - Start method, [174](#), [176](#)
 - Unity, [176](#)
- Enemy component, [419](#)
- Enemy Holder, [410](#), [411](#)
- Enemy script, [330](#)
 - method, [331](#)
 - protected keyword, [331](#)
 - variables, [330](#)
 - version, [332](#), [333](#)
- enum Mode, [376](#)
- EventSystem, Canvas, [368](#)
- Explicit conversion, [123](#)

F

- Fancy shooters, [305](#)
- File-type extension, [57](#)
- FiringTower script, [349](#), [350](#), [357](#)
- FixedUpdate built-in event
 - method, [554](#)
- FixedUpdate method, [511](#), [512](#), [525](#)
- Flying enemies, [329](#), [422](#)
- flyingEnemyPrefab, [411](#)
- FlyingEnemy script, [420](#)

flyinglevelInterval, 416

Force fields

adding velocity, 548, 549

position tool, 552

Rigidbody, 551

script class, 550

script creation, 545–547

set up, 547

Frames per second (FPS), 65, 78,
109, 110

Freezing time

declare method, 281

GUILayout.BeginArea call, 281

GUILayout.Button methods, 282

in-game pause menu, 283

movement/dashing methods, 280

rect constructor, 281

Time.deltaTime, 279

G

Game Lost Panel, 410, 411, 413

gameLostPanelInfoText, 410

GameObject.Find method, 277

GameObject.

GetComponentsInChildren<T>
method, 230

GameObjects

creating Cube, 18

dropdown menu, 18

Mesh Filter, 19

Mesh Renderer, 19

transform, 19

type, 373

Unity editor, 17

GameObject, setup

model holder, 150

transform component, 150, 151

Game Over screen, 423

Generic button, 368

Generic panel, 369

GetComponent method, 189, 344

GetNextCorner() method, 418

GetNextTarget() method, 354, 356

GetRandomPointWithin

method, 235

Getter property, 158

Goal script

collision detection matrix, 201

GameObject, creation, 202

prefab, 203

project setting window, 202

trigger collider, 204

using statement, 204

goldCost variable, 374

goldLastFrame, 378

GoToBuildMode() method, 413

Gravity method, 494

Ground enemies, 329, 422

GroundEnemy instances, 405

GroundEnemy movement, 417

groundEnemyPrefab

variable, 411, 420

GroundEnemy script, 404

GUILayout methods, 276

GUILayout.BeginArea, 284

GUILayout.Height, 284

GUILayout method, 361

H

Handles.color variable, 239

Hazard script, 17, 188–190

HD monitors, 363

Health bars, 443

HealthGainPerLevel variable, 416

INDEX

HideInInspector attributes, [334](#)
Hierarchy window, [14](#)
Highlighter cube, [377](#), [385](#)
Highlighter position, [386](#)
Hotkeys method, [463](#)
Hot plate, [435](#), [436](#), [438](#)

I

“If” statement, [62](#)
Inheritance, [439](#), [440](#)
 base class, [113](#)
 constructor chaining (*see* Constructor chaining)
 data, [113](#)
 declaring classes, [115–117](#)
 number value, [124](#)
 RPG, [113–115](#)
 subtypes/casting, [122–124](#)
 suffix, [124](#)
 doubles, [125](#)
 explicit conversion, [124](#)
 float value, [125](#)
 integer value, [125](#)
 unsigned version, [124](#)
 type checking, [126](#), [127](#)
 virtual methods, [127](#), [128](#)
Initializer, [219](#)
Input.GetKeyDown method, [101](#)
Input.GetMouseButton
 method, [323](#)
Inspector window, [14](#), [20](#)
Instance variables, [89](#)
Instantiate method, [197](#), [416](#)
Invoke method, [177](#), [180](#)
InvokeRepeating method, [414](#)
IsDashing property, [255](#)
Iterator, [219](#)

J

Joints
 chain link, [541](#)
 connected body, [542](#)
 Hovering Cube, [540](#)
 Rigidbody settings, [541](#)

K

KeyCode data type, [79](#), [80](#)
Kinematic Rigidbody, [197](#)

L

lastMousePosition, [378](#)
LateUpdate method, [462](#)
Layers
 Collision Matrix field, [329](#)
 dropdown, [327](#)
 Project Settings window, [328](#)
Leak method, [417](#)
Leak point, [397](#), [398](#)
LevelSelectUI script
 built-in method, [269](#)
 currentLoadOperation variable, [273](#)
 GUILayout.Button, [272](#)
 GUI methods, [270](#)
 level button, [276](#)
 main scene, [275](#)
 OnGUI method, [271](#), [272](#)
 scene build index, [270](#)
 Start method, [269](#)
 synchronous, [271](#)
 Update method, [273](#)
 using statement, [270](#)
Lives, [306](#)
Local position, [29](#)
Local variables, [89](#), [101](#)

LogInfo method, 99
 LogMyMessage method, 71
 LookRotation, 355

M

Materials, 152
 color field, 152
 color model, 154
 color popup window, 153
 components, 153
 hexadecimal value, 154
 HSV model, 154
 RGB model, 154
 Mathf.Max method, 170
 Mathf.Min method, 170
 maxDistance parameter, 441
 Max Slope, 403
 Mazing, 310
 Mesh Filter, 19
 Mesh Renderer, 19
 Method declaration, 70–72
 Methods
 empty GameObject, 66
 function, 64
 script, 66
 update, 65
 void, 65
 modelTrans reference, 247
 Modulus operator, 415
 Mouse-aimed camera
 first-person mode, 469, 470
 hotkeys, 463, 464
 OrbitPoint, 470–474
 player set up, 453, 454
 script setup, 456–462
 target GameObjects, 454, 455
 testing, 474

UpdateTargetRotation method, 464
 viewing player, 467
 xLookingDown/xLookingUp values, 468
 xRotation/yRotation variable, 465
 MouseDragMovement method, 322
 mouseDragSensitivity variable, 324
 Movement method, 486, 488–490
 Movement-related variables, 173
 movementSmoothing value, 321
 movementVelocity variable, 158, 159, 478
 MoveTowardsTarget method, 321
 Multiplier, 25

N

Navigation-related methods, 405
 Navigation window, 401, 402
 NavMesh asset, 403
 NavMesh.CalculatePath, 407
 NavMeshObstacle component,
 403, 404, 437
 NavMeshPath class, 405
 NavMeshPath instance, 442
 NavMeshPath stores, 418
 Nested prefabs, 48, 49
 Normalized vector, 479, 496

O

Object-oriented programming (OOP), 54
 Obstacle course design
 death/respawn, 145
 gameplay overview, 142, 143
 level, 145, 146
 mastering programming, 141
 player controls, 143, 144
 project setup, 147, 148
 scripts, 146

INDEX

OnClick event methods, [373](#), [388](#), [390](#)

OnColliderTouched method, [551](#)

OnSellTowerButtonClicked()
 method, [393](#)

OnSetup method, [335](#)

OnStageClicked method, [381](#), [386](#)

OnTriggerEnter method, [339](#), [551](#)

OnTriggerExit method, [344](#)

OnTriggerStay method, [551](#)

Operators, [73–75](#)

Overriding values

 asset, [44](#)

 components, [44](#), [45](#)

 GameObject, [46](#), [47](#)

 local position, [44](#)

 prefab instance, [44](#)

 prefab instance *vs.* asset, [46](#)

 root, [45](#)

 Skyscraper, [45](#)

 soldier, [44](#)

 transform component, [47](#)

P

Parameters, [68](#), [75](#)

Parenting, [29](#)

Pathfinding algorithm, [312](#), [442](#)

Pathfinding operation, [407](#)

Pathfinding setup, [401](#)

Patrollers, [209](#)

Patrol point

 arguments, [228](#)

 array (*see* Arrays)

 compiler error, [226](#)

 detect (*see* Detecting, Patrol points)

 indexer, [230](#)

 LookRotation, [228](#)

 private variables, [227](#)

 resembling, [209](#), [210](#)

 setting up (*see* Set up, Patrol point)

 sorting, [221–226](#)

 variable, [226](#)

PerformPathfinding method, [406](#)

Physics method, [375](#)

Physics playground design

 camera, [449](#)

 floating platforms, [451](#)

 force fields/jump pads, [451](#)

 player movement, [450](#)

 project setup, [451](#), [452](#)

 pushing/pulling, [450](#)

 swing, [451](#)

Physics.Raycast, [383](#), [384](#)

Pivot points

 Cube Base, [38](#)

 GameObject, [37](#), [38](#)

 skyscraper, [37](#), [38](#)

 tool handle position control, [36](#)

 transform tool, [36](#)

Platform script, [529](#)

PlatformDetector script, [529](#)

Platform movement

 big cube, [528](#)

 declaring variables, [523](#)

 FixedUpdate methods, [525](#), [526](#)

 GameObject, [527](#)

 Rigidbody, [519](#)

 Rigidbody.velocity, [522](#)

Play button

 active, [400](#)

 Build Mode heading, [409](#)

 locked, [408](#)

 Lock Panel, [400](#)

 normal/active, [400](#)

 setup, [409](#)

 variables, [410](#)

- Player Camera GameObject, 382
- Player movement
 - declaring variables, 155–158
 - GameObject (*see* GameObject, setup)
 - properties, 158, 159
 - velocity (*see* Velocity)
 - WASD/arrow keys, 149
- Player platforming, 529–532
- Player script, 373
 - build mode, 376
 - enum Mode, 376
- Player Settings, 301
 - fields, 302, 303
 - resolution/presentation, 301
 - splash screen, 303
- Play mode, 310
- PlayModeLogic method, 412
- Positions, 23
- PositionSellPanel method, 380, 389, 392
- Position tool, 22
- Prefabs
 - asset, 41, 43
 - editing, 42
 - making/placing, 41
 - variant, 49–51, 369
- Primitive collider, 543
- Private modifier, 91
- private void Movement() method, 254
- Programming languages, 53, 59
- Project, creation, 8
- Projectile class
 - arrow tower, 338
 - contents, 334
 - damage, 335
 - declaration, 337
 - error message, 337
 - projectileSeekPoint position, 336
 - speed variables, 334

- variables, 336
- Projectile layer, 329
- Projectile script, 190–194
- Project window, 12
- Protected keyword, 331
- Protected modifier, 91
- Public modifier, 91
- PullingAndPushing method, 515, 516
- Pulling/pushing
 - cursor drawing, 517, 518
 - Telekinesis script, 507–511

Q

- Quaternion, 169
- Quaternion.LookRotation
 - method, 170, 171

R

- Random.Range method, 247
- Range indicators, 444
- Raycast method, 384, 441
- Rect data type, 517
- Rect tool, 22
- RectTransform, 366
 - anchors, 366
 - circle icon, 366
- Rect transform tool, 263
- Reference type, 383
- Refund Text, 372
- Remove method, 394
- Rendering Mode field, 375
- Render Mode, 364
- Respawn method, 178, 180
- Retargeting, 233
- Rigidbody, 182, 546
- Rigidbody.AddForce method, 546, 551

INDEX

Rigidbody.velocity property, [554](#)

Roleplaying game (RPG)

- armor, [114](#)

- base class, [114](#)

- equipment, [114](#)

- fields, [113](#)

- terminology, [115](#)

rotationPerSecond variable, [107](#)

Rotation tool, [22](#)

S

Scale tool, [22](#), [27](#)

Scale/unit measurements, [24–26](#)

Scene flow

- build settings, [267](#), [269](#)

- LevelSelectUI script, [267](#)

- Player prefab, [268](#)

SceneManager.LoadScene method, [207](#)

Scene setup

- component, [522](#)

- hovering, [521](#)

- non-kinematic Rigidbody, [520](#)

- platform creation, [520](#)

Scene window, [13](#)

Screen Match Mode field, [365](#)

ScreenPointToRay method, [441](#)

Script Execution Order, [556](#)

Scripts, [13](#), [57–59](#), [103](#)

- attributes, [110](#), [111](#)

- class, [105](#)

- frames/seconds, [109](#), [110](#)

- GameObjects, [103](#)

- namespaces, [104](#)

- rotating transform, [107–109](#)

- using statement, [105](#)

- variables, [106](#)

- Vector3, [106](#)

- vector variable, [103](#)

- window tab, [104](#)

SeekingProjectile script class, [333](#), [338](#)

selectedBuildButtonColor, [378](#)

selectedBuildButtonImage, [378](#)

selectedTower, [378](#)

sellRefundText, [378](#)

Semicolons, [61](#)

SetActive method, [178](#), [374](#)

Setter property, [158](#)

Setup method, [336](#), [357](#)

Set up, Patrol point

- array of Transforms, [212](#)

- GameObject, [213](#)

- hierarchy, [216](#)

- index value, [213](#)

- model, [213](#), [215](#)

Shooting script, [194–197](#)

SimpleRotation script, [107](#)

Skyscraper GameObject, [33](#)

Slerp method, [169](#), [171](#), [246](#)

SpawnEnemy method, [415](#)

Spawn point, [397](#), [398](#)

Sphere Collider component, [341](#), [428](#)

Spike trap, [285](#)

- adding collisions, [295–297](#)

- copy-paste, [287](#)

- designing, [285](#)

- fully lower/raised, [286](#)

- GameObject, [286](#)

- material creation, [288](#), [289](#)

- one row spikes, [288](#)

- raising/lowering, [289](#), [290](#)

- repeating process, [292](#)

- script writing, [290](#), [291](#), [294](#), [295](#)

- sloppier, [287](#)

- vectors/Quaternion, [294](#)

stageLayerMask, [377](#)

StartLevel method, [414](#)
 StartLowering methods, [294](#)
 Start method, [331](#), [405](#), [406](#)
 StartRaising methods, [294](#), [296](#)
 Static members
 built-in methods, [101](#)
 instanced method, [99](#)
 Item class, [99](#)
 NumberOfInstances variable, [100](#)
 Start method, [100](#)
 Step Height, [403](#)
 Strong *vs.* weak typing, [55–57](#)
 Substring method, [231](#)
 Swing setup
 angular motion, [537](#)
 chain link, [536](#)
 compound colliders, [535](#)
 Configurable Joint component, [536](#)
 GameObjects, [533](#), [534](#)
 hovering cube, [539](#)
 platform, [540](#)
 primitive colliders, [535](#)
 Rigidbody, [535](#), [538](#)
 Synchronous operation, [276](#)
 Syntax, [53](#), [54](#)

T

TakeDamage method, [68](#)
 TargetDetection method, [513](#), [514](#)
 Targeter
 collider type, [340](#)
 creation, [340](#)
 distance, [348](#)
 enemies, [341](#), [344](#)
 looping, [344](#), [345](#)
 position, [347](#)
 reference, [348](#), [350](#)
 script, [346](#)
 towers, [343](#)
 Targeter.GetClosestEnemy method, [354](#)
 Telekinesis
 feature, [542](#)
 project settings window, [543](#)
 Rigidbody, [543](#)
 Teleportation, [305](#)
 Terrains, [555](#)
 Text component, [370](#)
 Textures, [152](#)
 3D movement
 ApplyVelocity method, [490–492](#), [495](#)
 CharacterController.Move method, [478](#)
 Gravity method, [494](#)
 Jumping method, [495](#)
 length, [479](#)
 magnitude, [479](#)
 maxGravity, [494](#)
 normalized vector, [479](#)
 player script, [481–487](#)
 project, [477](#)
 TransformDirection method, [480](#)
 VelocityLoss method, [493](#), [494](#)
 Time.timeScale, [284](#)
 Toolbar, [21](#)
 Tower
 arcing projectile (*see* Arcing projectile)
 barricades, [437](#)
 cannon (*see* Cannon tower prefab)
 hot plates, [435–437](#)
 instance, [385](#)
 maze set up, [437](#), [438](#)
 Tower defense games, [311](#)
 project creation, [313](#)
 project setup, [313](#)
 types, [311](#)
 towerPrefabToBuild, [378](#)

INDEX

- Tower Selling Panel, [371–373, 377, 389, 393](#)
- Tower.transform.position, [385](#)
- Transform.childCount member, [412](#)
- Transform class, [364](#)
- Transform component, [19](#)
- TransformDirection method, [480, 488](#)
- Transform tools, [21, 22, 24](#)
- Transform.TransformDirection method, [546](#)
- trans variable, [156](#)
- Trigger colliders, [182, 197](#)
- Typecast, [122](#)

U

- Unity
 - manual documentation, [135](#)
 - scripting API documentation, [136](#)
 - windows (*see* Windows)
- Unity Asset Store, [555](#)
- UnityEngine.UI, [376](#)
- Unity Hub, [1](#)
 - building projects, [3](#)
 - documentation, [4](#)
 - downloading button, [2, 3](#)
 - installation, [4](#)
 - license, [2](#)
 - modules, [3](#)
 - one-time setup, [2](#)
 - program, [9](#)
 - project (*see* Project, creation)
- Unity pathfinding, [309](#)
- Unity's documentation page, [304](#)
- Unity's UI system, [362](#)
 - Canvas, [363, 364](#)
 - elements, [363](#)
 - physical size, [365](#)
 - pixels, [365](#)
 - render elements, [364](#)

- scale mode, [365](#)
- scene window, [364](#)
- screen size, [364, 365](#)
- Tool Handle Rotation, [363](#)

- Unity users, [305](#)
- UpdateEnemyPath method, [394, 406, 407](#)
- Update method, [317, 337](#)
- Upgrade existing towers, [445](#)

V

- Value type, [383](#)
- Variables, [89](#)
- Variant asset, [50](#)
- Vector3.ClampMagnitude static
 - method, [489](#)
- Vector3 variable, [455](#)
- Velocity
 - Mathf.Min/Mathf.Max methods, [161](#)
 - script class block, [160](#)
 - speed backward, [162](#)
 - Time.deltaTime, [160](#)
 - Update method, [160](#)
 - up/down movement keys, [164](#)
 - VelocityGainPerSecond, [163](#)
 - X axis, [165, 166](#)
- VelocityGainPerSecond, [158](#)
- VelocityLossPerSecond, [158](#)
- Virtual keyword, [331](#)
- Virtual methods, [127, 128](#)
- void Update() method, [227](#)

W, X, Y

- WallIsNearby method, [499](#)
- Wall jumping
 - collisions, [497](#)
 - detection, [499–501](#)

- Update logic methods, [502–505](#)
 - variables, [497–499](#)
 - Walls
 - Floor plane, [199](#)
 - gizmo, [200, 201](#)
 - prefab, [199](#)
 - rect transform tool, [200](#)
 - tool handle rotation, [200](#)
 - Wanderer
 - Editor extension (*see* Editor extension)
 - region, [233–235](#)
 - scripts (*see* Wanderer scripts)
 - setup, [240, 241](#)
 - Wanderer scripts, [248](#)
 - enum, [241](#)
 - handling, state, [243, 244](#)
 - postRotationWaitTime, [243](#)
 - reacting, state, [245–247](#)
 - variables, [242](#)
 - Weapon/armor instance, [122](#)
 - Weapon constructor, [121](#)
 - Weapon variable, [124](#)
 - While statement, [62](#)
 - Windows
 - layout, [12](#)
 - tabs, [11](#)
 - World position, [29](#)
 - World scale, [35](#)
 - World *vs.* local
 - coordinates, [32](#)
- ## Z
- Zooming method, [325](#)